(2)

IDA PAPER P-2456

AN ASSESSMENT OF SOFTWARE PORTABILITY AND REUSABILITY FOR THE WAM PROGRAM

James P. Pennell, *Task Leader*

October 1990

Prepared for
Defense Communications Agency (DCA)

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and appropriate analytical methodology and that the results, conclusions and recommendations are properly supported by the material presented.

IDA PAPER P-2456

AN ASSESSMENT OF SOFTWARE PORTABILITY AND REUSABILITY FOR THE WAM PROGRAM

James P. Pennell, *Task Leader*

Cy D. Ardoin
John M. Boone
Karen D. Gordon
Robert J. Knapper
James Baldo
Bill R. Brykczynski
Deborah Heystek
Beth Springsteen
Craig A. Will

October 1990

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 89 C 0003
Task T-S5-771

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1990	Final

4. TITLE AND SUBTITLE
An Assessment of Software Portability and Reusability for the WAM Program

5. FUNDING NUMBERS
MDA 903 89 C 0003

T-S5-771

6. AUTHOR(S)
James P. Pennell, Cy D. Ardoin, James Baldo, John M. Boone, Bill R. Brykczynski, Karen D. Gordon, Deborah Heystek, Robert J. Knapper, Beth Springsteen, Craig A. Will

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses (IDA)
1801 N. Beauregard Street
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2456

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
JIEO/TVCF
Defense Information Systems Agency
Center for C3 Systems
3701 N. Fairfax Dr.
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution: 9 March 1992.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)

This paper provides the World Wide Military Command and Control System Automated Data Processing Modernization (WAM) program with the results of an examination of the topics of portability and reusability. This report will be used to assist the WAM program manager in determining the levels of portability and reusability that are needed in the program and in developing a plan to ensure that these levels are achieved. The portability discussion is limited to three services considered the most important for achieving applications portability: (1) the applications themselves, (2) their interface to the operating system, and (3) their interface to the data management system. This discussion was further focused on Ada applications portability, the Portable Operating System Interface for Computer Environments (POSIX), and Structured Query Language (SQL). The software reusability discussion focuses on benefits available now despite unresolved technical issues inhibiting wide-scale software reuse. The reuse of software offers the potential of increasing productivity in building parts of the system and increasing the quality of the system. These increases in productivity and quality can be expected to result in cost savings, reduced development time, higher system reliability, and other benefits. Recommendations are given for both topics.

14. SUBJECT TERMS	15. NUMBER OF PAGES
WAM; WWMCCS; Software Portability; Software Reusability; Ada Programming Language; POSIX; SQL.	158
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

EXECUTIVE SUMMARY

Portability and reusability are both important goals for the World Wide Military Command and Control System Automated Data Processing Modernization (WAM) program because its system architecture assumes that computers from different manufacturers will be connected together to form the system. The program strategy is designed to allow users to add new functions and to replace individual computers as needed. Users expect to achieve this flexibility without being made to purchase equipment or software from any particular supplier. They will be able to take advantage of previously purchased software or commercially available products.

Although both portability and reusability have been recognized as desirable features of software, much of the software purchased by Department of Defense (DoD) has been difficult to move to new computers. The WAM program is taking early action to identify factors that enhance these characteristics and to implement practices that promote design, development, and use of software that has better portability and reusability characteristics. As a first step in developing detailed guides for program policy and actions, this paper provides the WAM program with the results of an examination of the topics of portability and reusability.

PORTABILITY

There are three service areas where increased portability will be significant to the WAM program: application portability, operating system interfaces, and database interface. In the case of general application software, the use of a tightly regulated standard programming language and validated compilation system, such as with the Ada programming language, will enhance portability. Portability of the application software is further enhanced when applications are designed to operate over operating systems that present a standard interface. An operating system interface, the Portable Operating System Interface for Computer Environments (POSIX), is being developed specifically to promote portability. Similarly, in the case of a database language, the proper use of Standard Query Language (SQL) will increase the chances of portability.

However, simply selecting the appropriate standard is not sufficient to guarantee software portability. In the case of Ada, the standard allows the use of features which are inherently machine dependent [ANSI 1983, Chapter 13]. POSIX is not a single standard, but a collection of standards that describe variations of the "standard" interface. The SQL standard, while encouraging the use of a standard set of capabilities, does not provide every function that may be needed. All these standards are evolving. Thus, in addition to the selection of a standard, the software developer requires guidance in the

application of that standard to achieve the goal of portability.

Using standards such as Ada, POSIX, and SQL makes achieving a level of portability easier, but standards by themselves are not enough. Although Ada was designed to requirements which included portability, there are issues associated with Ada that affect portability of the software written in Ada.

- An understanding of the portability requirements of the application.
- A comprehensive understanding and familiarity with Ada.
- An understanding of the potential for misuse of the features of Ada.
- The selection or development of appropriate portability guidelines.
- The use of porting procedures and demonstrations.
- The use of independent validation and verification.

The operating system is the environment that the application program "sees" when it is executing. If the environment always appears the same, even when the computer changes or the operating system itself changes, then moving the software between computers will be easier. POSIX is a family of standards that has the potential for significantly facilitating applications portability. In addition to having IEEE support, it has broad U.S. Government and industry participation and support and is recommended in the WAM Decision Coordinating Paper (DCP) [DCA 1989, Appendix R]. It has been adopted as a key component of the Applications Portability Profile being developed by the National Institute of Standards and Technology (NIST).

However, the POSIX standardization effort is young and still under development. Further, there are three potential problem areas to be considered when using a POSIX open systems environment (OSE). Namely, POSIX allows for (1) optional features, (2) multi-semantic features, and (3) extensions.

The standard relational database language, SQL, while providing many features required for database management, does not provide significant support for schema manipulation, system tables, and interactive database users. As a result the ANSI committee developing the SQL standard expects to produce a new version of the language, SQL2, in 1992.

Portability Recommendations

Before the goal of portability can be met in the WAM program, several tasks must be successfully completed.

- Quantify portability. A measure of portability must be devised. One option is to use the source lines of code that must be changed when an application is

ported to indicate the relative percent of the software that is changed.

- Establish portability requirements. The requirement for portability must be understood if the software developers are to design the software in an intelligent and cost effective manner.
- Identify portability practices. The selection and appropriate use of available standards such as ANSI/MIL-STD-1815A-1083, IEEE Std. 1003.1-1988, and FIPS-PUB 127-1 will enhance the chances of developing portable software. These standards are evolving however. It is critical that the WAM program anticipate and prepare for these changes. One way to ensure timely knowledge and conformance with the standards is to participate in the standards efforts through balloting on proposed standards or membership in working groups.
- Monitor the development process for compliance. After selecting the appropriate standards, the goal of portable software is largely met through the use of appropriate software engineering practices such as the use and enforcement of a set of portability guidelines, an understanding of the issues surrounding portability, the use of a plan for achieving portability, and the use of automated tools to assist the software developers.
- Test the result. A portability demonstration must serve as an acceptance test for software developed for the WAM program. This demonstration will indicate whether the portability requirements established by the program office have been met. An independent V&V agent should also assess the portability of the delivered software.

REUSABILITY

Although major technical problems must be solved before the full potential of software reuse can be realized, some benefits can be obtained now. But software reuse is difficult because of such non-technical factors as organizational structures, financial disincentives, and lack of specific contractual mechanisms that allow and encourage reusable software.

The unresolved technical issues inhibiting wide-scale software reuse require further research. We see seven significant technical issues that are being (or should be) investigated, with progress in any areas likely to result in enhanced reuse capabilities:

- Improved methods for domain analysis.
- Improved indexing and retrieval systems for reuse libraries.
- Improved conceptual understanding and representations for reuse.
- Methods for raising the assurance that software performs as expected.
- Reuse methods that take into account the fact that software not only carries out

a functional task but does so with certain resource utilization characteristics

- Techniques for managing the increased number of parameters that are required for large components.
- Improved software tools for reuse.

Reusability Recommendations

Software reuse is not yet a mature technology. Although some benefits can be achieved on a small scale, reuse should not be attempted on a large scale at the present time. For example, a small-scale demonstration project designed to assess the applicability of reuse and the effectiveness of software tools in the technical and organizational environment of WAM should be initiated.

Before the benefits of software reuse can be realized, an analysis to determine the requirements for reuse in the WAM program must be conducted by the WAM program office. The results of this study will form the basis of the software design developed by the contractors.

The program office and WAM contractors should be aware of progress as research efforts to enhance software reuse capabilities continue. In addition, the work occurring in other DoD programs, such as Software Technology for Adaptable, Reliable Systems (STARS) and Strategic Defense Initiative Organization (SDIO), should be investigated for applicability to the WAM program.

While technical barriers to increased software reuse exist, contractual and legal problems continue to discourage reuse. Mechanisms to motivate and reward reuse must be developed. Questions about property rights and liability must be answered.

TABLE OF CONTENTS

Preface

The purpose of IDA Paper P-2456, *An Assessment of Portability and Reusability*, is to record substantive work done in a quick reaction study for the World Wide Military Command and Control System Automated Data Processing Modernization (WAM) program to follow in achieving appropriate levels of software portability and reusability.

This document fulfills a subtask of Task Order T-S5-771, *WAM Target Architecture*, which is to provide "a portability practices guideline that addresses programming language, operating system, and data query language services." It will be used to assist the WAM program manager in determining the levels of portability and reusability that are needed in the program and in developing a plan to ensure that these levels are achieved. The audience is the WAM program manager and principal deputies.

Peer review of this document was conducted by Dr. Richard Ivanetich, Dr. James Carlson, Mr. Terry Courtwright, Dr. Dennis Fife, Ms. Audrey Hook, and Dr. Robert Winner. Their contributions, and those of the editor, Ms. Katydean Price, are gratefully acknowledged.

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 PURPOSE

This paper provides the World Wide Military Command and Control System (WWMCCS) Automated Data Processing (ADP) Modernization (WAM) program with the preliminary results of an examination of the topics of portability and reusability. The study was begun in July 1990 and this report is a first step in developing detailed guides for program policy and actions.

1.2 BACKGROUND

Portability and reusability are characteristics of software that describe the ease of moving computer programs from one computer to another and of using parts of programs in new programs.[1]

Portability and reusability are both important goals for the WAM program because its system architecture assumes that computers from different manufacturers will be connected together to form the system. The program strategy is designed to allow users to add new functions and to replace individual computers as needed. Users expect to achieve this flexibility without being made to purchase equipment or software from any particular supplier. They will be able to take advantage of previously purchased software or commercially available products.

Although both portability and reusability have been recognized as desirable features for software, much of the software purchased by Department of Defense (DoD) has been difficult to move to new computers. The WAM program is taking early action to identify factors that enhance these characteristics and to implement practices that promote design, development, and use of software that has better portability and reusability.

1.3 SCOPE

This paper presents the findings of a preliminary examination of portability and reusability within WAM. A variety of services, each with its own interfaces, are needed

1. Portability is defined in the *IEEE Standard Glossary of Software Engineering Terminology* as "The ease with which software can be transferred from one computer system or environment to another" [IEEE 1983, 26]. The same source defines reusability as "The extent to which a module can be used in multiple applications"[IEEE 1983, 30].

to implement a distributed command and control information system (CCIS) such as WAM. We limit the present effort to just three[2] of the services that we believe to be among the most important for achieving applications portability: the applications themselves, their interface to the operating system, and their interface to the data management system. The portability discussion is further focused on Ada applications portability, the Portable Operating System Interface for Computer Environments (POSIX), and Structured Query Language (SQL).

Reusability, a well-established practice for math libraries and for packages that deal with common data structures, remains a topic of continuing research with regard to software modules in general. We address the general case and its attendant research issues.

1.4 APPROACH

We began work concurrently on three topics related to portability and on the subject of reusability.

The first topic discussed under portability is Ada. This section is concerned with designing and implementing applications programs that are themselves portable. Achieving better portability for applications programs written for the DoD was a goal of the higher order language studies that led to the development of Ada.

The information presented in the section on Ada portability is derived from a survey of open literature, a review of Ada portability guides, and IDA's contact with the Ada program for almost eight years.

The second topic discussed under portability is the operating system interface, POSIX. POSIX defines a standard interface between the applications and the operating system. This helps to ensure that software being transferred to new environments can take advantage of the same operating system interface found on the first computer. It is not intended to provide operating systems that are themselves portable.

The information contained in the POSIX discussion is based on a survey of open literature, a review of the IEEE 1003 group of standards and draft standards, and our ongoing research.

The third topic under portability is the interface to data management. A technique for promoting portability of software applications is the separation of data

2. We have identified seven classes of services necessary to support an open system architecture: programming language services, operating system services, data mangement services, data exchange services, network services, user interface services, and security services.

management from other applications functions. By keeping the data management function separate and providing a well-defined interface between it and other applications software, designers hope to encourage independence of applications from details of data storage. Different applications can share access to data and interactive users are not constrained to a single application when accessing data. SQL, which supports an interface to a relational database model, has been identified by the Defense Communications Agency as the preferred mechanism for achieving the desired interface between applications and data management [WWMCCS ADP TUG 1989, A-33].

The SQL discussion is based on a survey of open literature, a review of pertinent standards, and IDA's experience with the SQL standardization effort.

Less definitive work exists in the area of software reuse. There are no standards or guidelines on software reuse. However, a great deal of research on software reuse is occurring. The section on reuse identifies the state of practice and provides recommendations on how to move toward the goal of reusable software for the WAM program.

Achieving a greater level of reusability is a goal of several DoD programs, e.g., Software Technology for Adaptable Reliable Systems (STARS), Joint Integrated Avionics Working Group (JIAWG), and the Strategic Defense Initiative (SDI). Members of IDA staff are familiar with work in each program as well as independent research within IDA and elsewhere. The discussion and recommendations for reusability are derived from that knowledge.

2. PORTABILITY

Although any software can be evaluated in terms of portability, for this paper we consider the applications software (the programs written to carry out some function for the user.) Examples of application software include programs to prepare routine status reports, to aid staff officers in preparing part of an operation plan, or to help prepare a formatted message for release. Figure 1 represents a software application (dashed rectangle) as a set of packages (solid rectangles). In this figure, some of the packages (the shaded ones) might require modification of the code if the application is moved to a new computer. If software is designed according to accepted principles, then the changes should be (1) known about before hand, (2) as few as possible, (3) isolated in just a few packages, and (4) not capable of causing changes in the operation of other packages. The application itself interacts with the host computer through its interface to the operating system and, in some cases, through its interface to the data management system. The application interfaces to applications that are executing on remote computers via a communication system interface. These interfaces form part of the system architecture.

The advantages of portable software include increased programmer productivity and increased software reliability, flexibility, and maintainability. However, the benefits of portable software are only realized over time. As hardware technology continues to advance, the availability of greater performance, reliability, or functionality may cause the original hardware to be replaced. Software initially developed to be highly portable may accommodate the changes and continue to be used.

We expect that the application software written for WAM will be installed and executed on different computers and on different versions of operating systems. For this reason, portability is an important characteristic of software developed or purchased for the WAM program.

Three service areas where increased portability will be significant to the WAM program are the application software, the operating system interface, and the database interface. In the case of general application software, the use of a tightly regulated standard programming language and validated compilation system, such as with the Ada programming language, will enhance portability. Portability of the applications software, is further enhanced when applications are designed to operate over operating systems that present a standard interface. An operating system interface, the Portable Operating

System Interface for Computer Environments (POSIX), is being developed specifically to promote portability. The WAM program has specified POSIX as its operating system interface for the workstations that have been purchased [DCA 1989, Appendix R]. Similarly, in the case of a database language, the proper use of Standard Query Language (SQL) will increase the chances of portability.

However, simply selecting the appropriate standard is not sufficient to guarantee software portability. In the case of Ada, the standard allows the use of features which are inherently machine dependent [ANSI 1983, Chapter 13]. POSIX is not a single standard, but a collection of standards that describe variations of the "standard" interface. The SQL standard, while encouraging the use of a standard set of capabilities, does not provide every function that may be needed. The SQL standard is still evolving to meet the needs of database application developers. Thus, in addition to the selection of a standard, the software developer requires some guidance in the application of that standard to achieve the goal of portability. The following three sections address the issues of standards and guidance in the development of application software that interacts with operating and database applications through standard interfaces.

2.1 APPLICATIONS PORTABILITY THROUGH ADA

2.1.1 Introduction

Some of the features that characterize portable software are common across programs written in any language. For instance, reliance on machine-dependent features or vendor-supplied math packages will result in non-portable code regardless of the programming language selected for the implementation. Similarly, intelligent use of a programming language standard, avoidance of host-specific features, and adoption of programming style guidelines will tend to enhance portability.

The Ada programming language has been mandated by DoD Directives 3405.1 and 3405.2 for use in developing all mission-critical software [Taft 1987a, 1987b]. A majority of the software developed for the WAM program will therefore be written in Ada. For that reason, this section will address the use of the Ada programming language to improve the portability of software.

2.1.2 Background

In the mid-1970s, the DoD set up a High Order Language Working Group (HOLWG) as part of its Common High Order Language program. The goal of the program was to establish a single high order computer programming language with which to develop mission-critical embedded computer systems. The HOLWG was tasked with formulating DoD requirements for high order languages and evaluating existing languages

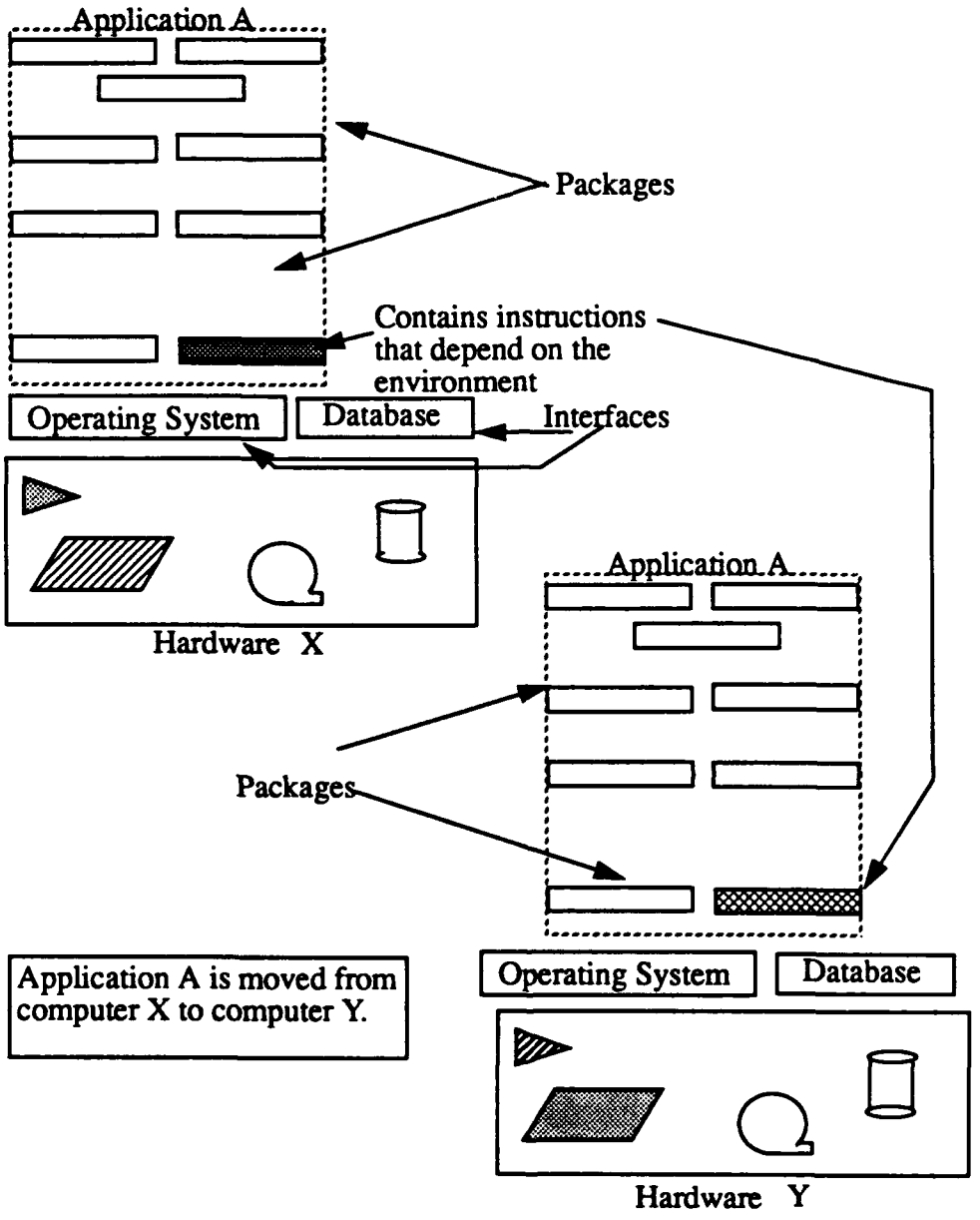

Figure 1. Example of Portable Application Software

against those requirements. By 1978, the HOLWG had developed a set of requirements, referred to as STEELMAN [DOD 1978], and had determined that none of the existing military and commercial programming languages met those requirements sufficiently. However, the HOLWG also determined that a single language that would essentially meet all of the requirements was feasible. The Ada Programming Language Standard [ANSI 1983] resulted from the evolution of a DoD-sponsored design which attempted to satisfy STEELMAN.

One requirement called out in STEELMAN was for machine independence [DOD 1978, 4]. Machine independence is nearly synonymous with portability as defined by [IEEE 1983]; therefore the Ada language was designed with portability in mind. The high-level nature of Ada, especially its separation of specifications from implementation details and the package facility for encapsulating dependencies, reflects Ada's attempt at the programming language level to promote portability.

2.1.3 Portability Issues

Although Ada was designed to requirements which included portability, there are issues associated with Ada that affect the portability of software written in Ada.

- Determining the requirements for portability
- Comprehensive understanding and familiarity with Ada
- What guidelines should be used
- Porting procedures and porting demonstrations
- Independent validation and verification

Portability requirements analysis should specifically consider which components must be developed to be portable. The cost associated with the development of portable Ada software is generally believed to be higher than for the development of non-portable Ada software. Substantial savings may be gained over time if that software can be ported to a new environment rather than being redeveloped. However, the savings can only occur when there is a need to port the software. Portable software may also be less efficient during execution or require additional memory use than does non-portable software. In general, there are always tradeoffs to be considered when optimizing software for a specific trait such as portability. Thus, an evaluation of the requirement for portable software must occur before any contracting decisions can be made. A level of portability, such as 80% portable, is meaningless if the wrong modules are portable. The criteria for deciding which software modules need to be portable may include life expectancy of the module, commonality across the system, initial cost of development, or behavioral constraints such as performance. Any portability requirements analysis must consider the benefits of developing a portable software module versus the cost associated with that

development effort.

Creating portable Ada software can not be done without a comprehensive understanding and familiarity with Ada. The potential exists for misuse of the features and facilities provided by Ada to assist in minimizing machine-dependencies. For example, a program relying upon pre-defined numeric types for their bounds, size, or accuracy, will not minimize the machine-dependence of that program. Compiler conformity with the Ada standard, although rigorously tested by DoD, does not mean that Ada source code processing is identical between compilers. One specific instance of this that we have encountered through our experience is with the nesting of Ada generic units. The nesting of generic units is allowed by the standard, but not thoroughly tested against in conformity testing. As a result, some validated compilers handle nested generic units properly while at least one does not.

Once the portability requirements and the documentation and demonstration mechanisms have been determined, the software developer must select or develop portability guidelines to facilitate achieving those requirements. The process of selecting or developing guidelines is important as the guidelines will drive the entire software development process. The guidelines will be used by developers in making all implementation decisions and by software development managers in approving the results of the development effort. The use of inappropriate or inferior guidelines may be as bad as using no guidelines. The cost to develop a set of guidelines will be much higher than using an existing set of guidelines. However, the cost of using an inappropriate guideline must be considered. Having selected or developed a set of portability guidelines, it is critical that the adherence to the guidelines is enforced. This may involve the use of an automated tool specifically developed to check for guideline conformance.

Blind acceptance of Ada software as portable simply because it is written in Ada is not wise. Applications software can be designed for initial operation on several different computers yet it may later be difficult to move to some new computer. Therefore, if customers anticipate a need to move software to new, but unspecified, computers without having to rely on the original developers, then they should insist that software include complete porting instructions[3]. In the case of WAM, for example, software might include a porting instructions for moving the application between WAM hosts and alternative but equivalent platforms. The accuracy of the instructions can be tested by having an independent agent (perhaps an Independent Verification and Validation (IV&V) contractor)

3. The porting instructions allow people who did not participate in the original development to move the application to a new environment. Complete instructions, however, will not guarantee that anyone can move the application with equal ease. People who are more familiar with the application and the environments involved will probably have an easier time moving software than will people who are less qualified.

actually carry out the move using the instructions provided. Requiring porting procedures and demonstrating those procedures gives more assurance that the software meets the portability specifications. Selecting computers to use for portability demonstrations is a challenging task, but differences in computer word size, register architecture, instruction set, memory hierarchy, operating system, compiler, and database management system vendor tend to make the demonstration more robust.

If specific host porting requirements cannot be defined for the WAM program, acceptance testing can be based upon a weaker criteria, i.e., a general assessment of the portability of the software as determined by code inspection. This inspection could provide an analysis of the software for attributes which enhance portability. Although no standard set of attributes for portability in Ada exist, the existing portability guidelines can be used to derive these attributes.

The final issue of importance is IV&V of the software to the portability requirements. As discussed in preceding paragraphs, the use of Ada does not guarantee portability. Although it requires additional funds and other resources, IV&V of the porting procedures and the demonstration of requirements satisfaction is strongly recommended.

For the remainder of the discussion of achieving portability using Ada, we concentrate our efforts on the issue of selecting suitable portability guidelines.

2.1.4 Portability Guides

Guides have been written on the subject of Ada software portability. These guides generally define what portability is, why it is important, and what some of the issues surrounding portability are. Some of the recommendations contained in these guides are actually style guidance and simply establish a convention for the ways things are done.

Style guidelines by themselves are not sufficient to guarantee portability; however style guidance is important. By establishing conventions, the opportunity exists for using automated tools to assist the developer with the more mundane aspects of building software. For example, templates can be generated to speed the development process and minimize clerical errors. Alternatively, tools may be built to check that the conventions have been followed. While stylistic guidance will increase programmer portability as well as provide the opportunities for other benefits, stylistic aspects of the use of the Ada programming language will not be addressed in this paper. Interested readers should refer to [Nissen 1984] or [SPC 1989] for more information on Ada style.[4]

4. Two copies of SPC's *Ada Quality and Style* are provided with this paper.

As part of our research into Ada software portability, we reviewed[5] the available guides that have been written for Ada:

- Nissen, *Portability and Style in Ada*[6] [Nissen 1984]
- SofTech, *Ada Portability Guidelines* [SofTech 1984]
- Martin Marietta, *Software Engineering Guidelines for Portability and Reusability* [Martin Marietta 1989]
- SPC, *Ada Quality and Style* [SPC 1989]
- Griest, *Limitations on the Portability of Real-Time Ada Programs* [Griest 1989]

We found that in general, the guides may be characterized by approach, structure, level of technical detail, domain for which they were written, and age. Each characterization is discussed in the following sections.

2.1.4.1 Approach

The approaches taken in the development of the guides varied widely. The first Ada portability guide [Nissen 1984] was developed during the Ada language standardization effort. The goal was to encourage the development of portable Ada code from the outset. Some guides that followed used this first guide as the basis for their work while offering specific enhancements. In some cases, the guides were written to provide the software developer with a method to perform tradeoff analyses in determining whether to apply or disregard specific guidance. A result of this approach is that developers have latitude in implementing their design decisions. The developers must consciously analyze the risks and benefits associated with any decision. Thus the rationale for any decision can be evaluated and verified. In other cases, the guidelines were simply stated as suggestions, recommendations, or mandatory rules to be followed with no explicit justifications presented. Using these guides the software developer must blindly follow the guidance offered. However, this may be appropriate in some instances and will provide the benefit of allowing automated tools to check for rule conformance.

One set of guidelines resulted from a single porting effort and thus describes one instance of porting problems and their solutions [Martin Marietta 1989]. Another classifies software development personnel into groups of inexperienced developers, experienced developers, and development managers [SPC 1989]. These guidelines are aimed at providing specific assistance to each group. The approach taken in the development of portability guidelines may be to provide guidance for the development of portable code

5. The reviews are located in Appendix D.

6. This guide is no longer in print and will be difficult to obtain.

for a specific domain [Griest 1989]. The approach taken in the development of a set of guidelines may be considered when selecting an appropriate set of guidelines for a particular project or group of developers. The experience of the developers, constraints or domain of the project, and availability of automated tools will all influence the decision of which set of guidelines is most suitable.

2.1.4.2 Structure

The structure of the guidelines reviewed varied less widely than the approach taken in the development of those guidelines. However, guideline structure may also be used as a criterion in the selection of an appropriate set of guidelines for a project. Several guidelines were written to conform to the organization of the Ada standard. These guidelines provide a systematic treatment of Ada language constructs with the guidelines numbered to correspond to the relevant chapters and sections of the Ada standard. Another guideline was organized around issues impacting portability. For instance, a guideline section on concurrency enumerated the guidance for using Ada features to achieve concurrent, portable Ada code. Because a software developer might need guidance on a particular feature of the Ada language, this set of guidelines was also cross referenced to the Ada standard.

2.1.4.3 Level of Technical Detail

All of the guidelines reviewed contained some guidance that is more appropriately classified as style guidance. That is, the guidance consists of discussions on good software engineering practices. The reason for this may be that experience in porting Ada code has been limited. Style guidance is important but insufficient to guarantee portability.

2.1.4.4 Domain

Most of the guidelines reviewed were aimed at the development of general purpose Ada applications. Only one was limited to the domain of real-time embedded systems. It is to be expected that the more specialized the problem becomes, the more constrained the guidelines will need to be.

2.1.4.5 Age

The first guideline for the development of portable Ada applications was written while the Ada language standardization process was occurring. Since that time Ada compliers have proliferated making the use of Ada on a wide range of platforms possible. Developers continue to gain substantial experience with Ada. It is through this experience that better guidance for the generation of portable Ada code will result. Features of

the Ada language have been discussed and their use clarified through the language revision process. Some guidance initially proposed has now been changed or made obsolete. This will continue to occur. Thus, any software developer must remain aware of the experience of other developers in the field, as well as information derivable from the Ada Issues or ISO Ada Uniformity Rapporteur Group (URG) Issues.

2.2 APPLICATIONS PORTABILITY THROUGH POSIX

A software application is executed under the control of an operating system and requests other system resources (files, memory, input/output devices, etc.) by making calls to the operating system. In a sense, the operating system is the environment that the application program "sees" when it is executing. If the environment always appears the same, even when the computer changes or the operating system itself changes, then moving software between computers will be easier.

In this section, we put forward POSIX as a family of standards that has the potential for significantly facilitating application portability. We begin by giving an overview of POSIX, a snapshot summary of POSIX as it exists today, and a discussion of the significance of POSIX as an open system standard. We then focus on portability considerations. In particular, we describe the POSIX approach to application portability, and we point out some potential problems with this approach.

2.2.1 Background

POSIX has evolved into a term with a broad and still-increasing scope. It is used to refer to the standards being developed by IEEE Project P1003, which is sponsored by the Technical Committee on Operating Systems of the IEEE Computer Society. The term POSIX is also used to refer to P1003 itself, as well as to the collection of working groups that exist under P1003. In addition, it is used as an umbrella term to encompass not only P1003, but also some closely related IEEE standards projects (e.g., P1201 on windows, P1237 on remote procedure call (RPC), and P1238 on the File Transfer, Access and Management (FTAM) Protocol).

IEEE Project P1003 consists of a family of working groups (see Table 1). The P1003 working groups are defining interface standards based on UNIX®. All of the P1003 standards are intended to facilitate application portability at the source code level. While UNIX has become the operating system of choice on a large number of widely varying hardware platforms, its proliferation of versions in fact impedes application portability. The P1003 working groups are chartered to remedy this situation by defining a *standard* operating system interface and environment based on UNIX.

® UNIX is a registered trademark of AT&T Bell Laboratories.

TABLE 1. POSIX Working Groups and Standards

POSIX Overview				
Working Group	**Subject**	**Balloting**	**Expected Approval**	**Related Standards**
FIPS 151-1	POSIX	n/a	**Approved 3/90**	IEEE P1003.1-X (POSIX)
P1003.0	Guidance			
P1003.1	Systems Services	n/a	**Approved 8/88**	ANSI (1989), ISO 9945-1
(1989)				
P1003.2	Shell & Utilities	1990	1990	ISO DP 9945-2 (expected 1990)
P1003.2a	User Portability Extensions	8/90		
P1003.3	Test Methods, generic	2/90	late 1990	
P1003.3.1	Test Methods, system interfaces	2/90	late 1990	
P1003.3.2	Test Methods, shell & utilities	early 1992	late 1992	
P1003.4	Real-Time Extentions	5/90	mid 1991	
P1003.4a	Threads	8/90		
P1003.4b	Language Independence	12/90	late 1991	
P1003.4c	Extensions to P1003.4	late 1991		
P1003.5	Ada Language Binding	8/90	early 1991	
P1003.6	Security	5/91	early 1992	
P1003.7	System Administration	early 1992	1993	
P1003.8	Transparent File Access	mid 1992		
P1003.9	FORTRAN Language Binding	8/90		ISO DIS 8806, X3.9-1978
P1003.10	Supercomputing AEP	late 1990		
P1003.11	Transaction Processing AEP	mid 1991		
P1003.12	Protocol Independence	1993		
P1003.XX	Name Space / Directory Services		[1]	
P1003.13	Real-Time AEP	early 1991		
P1003.XX	Traditional System AEP	mid 1991		
P1003.14	Multiprocessing AEP	mid 1991		
P1003.15	Supercomputing	7/91		
P1003.X	C Language Binding			
P1201.1	X Windows - Applications API			
P1201.2	User Interface Driveability			
P1201.3	User Interface Management			
P1201.4	X Windows - Library API		early 1991	
P1224	X.400 Mail Services		[2]	
P1237	Remote Procedure Calls API	mid 1992	early 1993	
P1238.1	FTAM - Common OSI API	early 1992		
P1238.2	FTAM API	early 1992		ISO 8571

[1] - PAR has been withdrawn.
[2] - PAR may be withdrawn.

PAR - Project Authorization Request
FTAM - File Transfer, Access and Management
AEP - Application Environment Profile
API - Application Portability Interface

The first P1003 working group, P1003.1, has produced a standard now known as IEEE Std 1003.1-1988. IEEE Std 1003.1-1988 defines the interfaces to system services, including process management, signals, time services, file management, pipes, file I/O, and terminal device management. IEEE Std 1003.1-1988, in the UNIX tradition, is oriented toward the interactive multi-user application domain. Using IEEE Std 1003.1-1988 as a baseline, other P1003 working groups are extending application portability to additional application domains, as noted in the following section.

2.2.2 Snapshot Summary of POSIX Working Groups and Standards

The set of working groups that make up the POSIX family is evolving; additional working groups may be formed, and support for some existing groups may be withdrawn. Part of the purpose of the P1003.0 working group is to facilitate the coordination of the individual working groups. The purpose of this section is to give a snapshot description of the current POSIX family. In the following sections, we briefly describe the POSIX working groups and standards under eight headings:

- Guidance
- System services
- Utilities
- Language bindings
- Distributed system services
- Windowing
- Conformance
- Profiles

These headings are derived from the *POSIX Tracking Report* [Digital 1990]. It should be noted that the headings do not represent an official classification scheme. They simply provide a convenient framework in which to discuss the numerous working groups and standards that fall under the POSIX umbrella. Although most of the POSIX effort is intended to produce standards that will enhance applications portability, only the last topic, Profiles, explicitly addresses it.

Further information on the POSIX working groups and standards can be obtained from the publications listed in the bibliography or from the chairs of the working groups (see Appendix A). POSIX drafts can be obtained from the IEEE Computer Society by contacting Lisa Granoien at (202) 371-0101. A complimentary subscription to *POSIX Tracking Report* can be obtained by contacting Kate Comiskey at (603) 881-1873.

2.2.2.1 Guidance

Guidance for the POSIX effort is offered by working group P1003.0. The P1003.0 draft standard is intended as a centralized document which provides a detailed description for each standard in the POSIX family. The scope for each standard is included as well as a cross reference to related standardization activity by other organizations. A major goal of the working group for this standard is to identify gaps in the POSIX standards structure, and to create new working groups to address those areas. This document is drawing increasing attention from user organizations such as the National Aeronautics and Space Administration (NASA) and the Navy's Next Generation Computer Resources (NGCR) Program.

2.2.2.2 System Services

System services are being defined by three POSIX working groups:

- P1003.1 Systems Services and C Language Binding. This standard defines an interface primarily for low-level system routines, but also includes some higher-level (library) interfaces. It addresses process management, signals, time services, file management, pipes, file I/O, and terminal device management. It is oriented toward the interactive multi-user application domain and to centralized computer architectures. Current efforts are focused on removing the C language dependencies from the standard. A language independent specification is being developed. It will be used as a baseline for developing language bindings, especially for languages other than C (e.g., Ada, FORTRAN).
- P1003.4 Real-Time Extensions. This standard will address issues which are of a particular concern to real-time applications developers, who generally consider a traditional UNIX environment to be unacceptable for fielding these applications. In real-time systems, resources must be managed so that time-critical application functions can control their response time, possibly resulting in delay or even starvation for non-time-critical application functions. Therefore, the P1003.4 Working Group has focused its initial efforts on defining application interfaces to the functional areas which impact resource management (e.g., priority scheduling, real-time files, and process memory locking).
- P1003.6 Security. This standard will address computer security, which is generally considered not very rigorous in a traditional UNIX environment. It will define interfaces to security services and mechanisms. The working group's basis for consideration of security issues is the DoD 5200.28-STD Trusted Computer Security Evaluation Criteria (TCSEC or the "Orange Book") [DOD 1985]. Currently, the major features specified by the standard include

discretionary and mandatory access controls, audit mechanisms, privilege mechanisms, and information labels (added after the April 1990 meeting).

2.2.2.3 Utilities

Three working groups are addressing the topic of utilities:

- P1003.2 Shell and Utilities. The P1003.2 standard will define a standard programmatic interface to utilities which are commonly provided under UNIX. These utilities and the command interpretor (shell) are particularly popular features of UNIX systems, which increases the importance of this particular standard.
- P1003.7 System Administration. This group is addressing topics such as backup, recovery, system startup, system shutdown, clock daemons, print spooling, file management, and system code messaging.
- P1003.15 Supercomputing Batch Environment Extension. This standard will be developed by the same working group as the P1003.10 (Supercomputing Application Environment Profile (AEP)) working group. It will define facilities that provide a network queuing and batch system in a POSIX environment.

2.2.2.4 Language Bindings

A large effort is being made to remove the C language dependencies from the POSIX family base. This effort is being driven in part by the desire to carry POSIX into the international standards arena. The current plan is to supplement the language-independent standards base with interface definitions for specific languages. This is a difficult task since the origin of the POSIX family, UNIX, has matured under the influence of the C language syntax and semantics. The documents reviewed in this section represent the interface definitions which have currently been specified.

At this point, working groups have been established to define language bindings for two languages other than C:

- P1003.5 Ada Bindings. The P1003.5 will not be a standard as such, but rather a supplemental document to the P1003.1 (Systems Services) standard. As noted above, the P1003.1 standard is being revised to remove language-specific dependencies. As the Ada interface description will be of significant interest to U.S. federal employees and contractors, this document will figure heavily into the WWMCCS modernization. After completing the binding to P1003.1, the P1003.5 working group intends to define bindings to the P1003.4 real-time extensions.
- P1003.9 FORTRAN Bindings. This standard will initially define

FORTRAN 77 bindings to the POSIX standards.

2.2.2.5 Distributed System Services

The POSIX groups working on distributed services are trying to define Application Program Interfaces (APIs). At one time, the work to extend POSIX to a distributed computing environment was concentrated in one working group, P1003.8. However, the effort has evolved into several working groups:

- P1003.8 Transparent File Access API. This group is defining an API for a transparent file access facility similar in functionality to Sun Network File System (NFS)®.
- P1003.12 Protocol Independent Interfaces (PII) API. This group is defining an API for a network independent data transport capability.
- P1003.XX Namespace & Directory Services (NS/DS). This standard will be based on CCITT Recommendation X.500 [ISO 1988a].
- P1224 X.400 Mail Services API. This group plans to define an API for a mail service based on CCITT Recommendation X.400. However, due to lack of critical mass for participation in the working group, IEEE support may be withdrawn.
- P1237 Remote Procedure Call (RPC) API. This group is defining an API for a remote procedure call facility, which enables procedure calls to be made across a data communication network.
- P1238 File Transfer, Access and Management (FTAM). This group is defining an API for FTAM, ISO 8571 [ISO 1988b].

2.2.2.6 Windowing

Another IEEE Project, P1201, is defining a standard windowing interface based on X Windows. The P1201 interface is being designed to work with any operating system. It is not dependent on POSIX, although it is anticipated that it will often be used in conjunction with POSIX. It is commonly placed under the POSIX umbrella, and its working group meets jointly with P1003.

2.2.2.7 Conformance

Working group P1003.3 is chartered to develop test methods for measuring conformance to POSIX. The P1003.3 standard will define a uniform way of testings systems for conformance to the P1003.1 (Systems Services) standard. For this reason, the working group has closely monitored the development of the P1003.1 standard to determine and

® Sun Net File System is a registered trademark of Sun Microsystems, Inc.

document all assertions about system behavior. The P1003.3 working group has come to realize that it cannot address all the POSIX standards by itself. The current plan is for individual working groups to develop test assertions for their own standards.

As of April 1990, the focus of this working group has broadened into three major divisions. The P1003.3 standard will address generic test methods which defines how to write assertions and test methods for the other standards. The P1003.3 document is expected to be completed near the end of 1990. Two other documents, P1003.3.1 and P1003.3.2, will specify the test methods for the P1003.1 (Systems Services) and P1003.2 (Shell and Utilities) standards, respectively. Since the P1003.3.1 document encompasses revisions to the Systems Services standard (P1003.1a), it is not expected before early 1991. The P1003.3.2 effort is still in an early stage of development.

2.2.2.8 Application Environment Profiles

Profiles are being developed with groups of experts in the application area who identify existing application standards and focus on applying and extending POSIX to meet their specific needs. If a standard has many options, profile developers select the most appropriate options to meet their needs. The profile approach has also been successful for identifying areas where there are no existing standards to satisfy the application's needs. In which case, several approaches can be taken to ensure these needs are met. The Application Environment Profiles (AEP) groups can develop the standard themselves, approach the base standards groups, or form a new standards group [Isaak 1990, 67-70]. Although these profiles may provide useful standards in the WAM context, there are portability issues to be addressed.

First, a POSIX AEP is a subset of POSIX OSE, plus an arbitrary collection of options, parameters and extensions. As a result, the use of these profiles will restrict the portability of applications to those environments (domains) that are supported by the POSIX AEP. For example, files may not be supported in the Real-Time AEP, and semaphores may not be supported in the Transaction Processing AEP. These factors must be considered before choosing a POSIX AEP. An improper selection will require the use of non-portable options and extensions, and this will have an adverse affect on the portability of applications. Second, each POSIX AEP may contain optional features, multi-semantic features, and extensions, and their use must be controlled in a similar fashion to the non-portable features of POSIX OSE.

2.2.3 Significance of POSIX

POSIX is the key open system standardization effort in the area of operating systems. In addition to having IEEE support, it has broad U.S. Government participation

and support. It has been adopted as a key component of the Applications Portability Profile being developed by the National Institute of Standards & Technology (NIST). Draft 12 of the P1003.1 standard was adopted as Federal Information Processing Standard (FIPS) 151 in September 1988. Draft 13, which became IEEE Std 1003.1-1988, was adopted as FIPS 151-1 in March 1990. POSIX has been selected by NASA for use in its development of the Space Station Freedom. Most recently, in April 1990, POSIX was selected by the Navy Next Generation Computer Resources (NGCR) Program as the baseline on which to build the NGCR operating system interface standard [OSSWG 1990]. POSIX is now being carried forward into the international standards arena as well. IEEE Std 1003.1-1988 (plus 1003.1a, which makes minor revisions to the 1988 IEEE standard) is expected to be adopted as ISO 9945-1 in late 1990.

POSIX also has broad industry participation and support. All major computer vendors are at least monitoring the POSIX standardization effort by attending meetings and reviewing draft standards, and many are making substantive contributions. In addition, prominent industry consortia, such as X/OPEN, the Open Software Foundation (OSF), and UNIX International[7] are participating in the POSIX standardization effort and are incorporating POSIX into their plans for their own UNIX standardization efforts. The POSIX standards are expected to become both widely available and widely utilized.

Open system standards are important because they form the foundation for open system environments. As defined in [NIST 1990, 2], "open system environments" (OSEs) are ones:

- "that are based upon an architectural framework which allows an extensible collection of capabilities to be defined,
- in which capabilities are defined in terms of non-proprietary specifications that are available to any vendor for use in developing commercial products, and
- whose evolution is controlled by a consensus-based process for decisions regarding capability definitions, specifications, and other issued related to the computing environment."

As described in [NIST 1990, 3], the POSIX-based OSE is based upon a framework which divides services (provided by a computing system to an application) into six categories: operating system services, programming services, data management services, data interchange services, network services, and user interface services. The POSIX standards primarily address the operating system services. In the case of the POSIX-based OSE, standards compatible with the POSIX standards are used to address the

7. These three organizations and their interrelationships are described in [Grindley 1989]. Essentially, they are alliances of computer and software vendors with vested interests in the standardization of UNIX.

other five categories of services.

Ideally, an open system environment would be defined in terms of international standards, since consensus on as broad a scale as possible is desirable. However, as noted in [NIST 1990, 3], the set of international standards is not rich enough at this time to enable the definition of a complete, consistent open system environment. Therefore, in defining the POSIX OSE, standards have to be selected from many forums. The P1003.0 draft standard [IEEE 1989, 40] gives a list showing the precedence of standards for inclusion in the POSIX OSE. At the top of the list are "Approved standards developed by accredited international bodies," and at the bottom of the list are "Approved standards developed by non-accredited national standards bodies using a closed forum."

Many benefits can be accrued from the adoption of open system environments and open system standards. These benefits include (1) portability of applications, (2) connectivity and interoperability of computer systems and products, (3) protection of software investment, due to the portability of software to new computer systems that conform the open system standards, and (4) encouragement of commercial, off-the-shelf (COTS) acquisitions. COTS acquisitions in turn offer advantages in terms of "timeliness, cost, reliability, completeness of documentation, and training [DSB 1987, 3]." Moreover, COTS acquisitions can significantly reduce the burden that must be borne by the customer (such as the WAM Program) in the area of life cycle support.

2.2.4 POSIX's Vision for Portability

POSIX working groups have been involved in defining a set of standard specifications which comprise the operating system kernel and utilities. The base standards (e.g., system services, utilities, language bindings, and distributed system services) try to specify functionality and interfaces to satisfy a variety of diverse interests. For example, P1003.4 Real-Time Extensions for Portable Operating Systems must satisfy the needs of embedded real-time, "soft" real-time, transaction processing, and reliable database applications, to name a few [Naecher 1990, 46-51]. For this reason, the final standards are rather large and unwieldy containing all the options necessary to satisfy each application area. To minimize this problem, POSIX has formed groups to define AEPs that support portability in a specific domain [IEEE 1990, 46-51].

Profiles are being developed with groups of experts in the application area who identify existing application standards and focus on applying and extending POSIX to meet their specific needs. If a standard has many options, profile developers select the most appropriate options to meet their needs. The profile approach has also been successful for identifying areas where there are no existing standards to satisfy the application's needs. In which case, several approaches can be taken to ensure these needs are

met. The AEP groups can develop the standard themselves, approach the base standards groups, or form a new standards group [Isaak 1990, 67-70].

2.2.4.1 Role of Application Environment Profiles

An AEP is a collection of interface standards tailored to a particular application domain. The idea is that applications would be implemented in accordance with an AEP for a specific application domain. Then, applications conforming to a given AEP would be portable across systems that implement the same AEP.

The purpose of AEPs is to help specify systems that can be built or procured so that the procurement office, developers, users, and platform suppliers can communicate their needs in an unambiguous manner. AEPs are meant to simplify the software developer's task of identifying relevant standards to ensure the application is portable. System purchasers can avoid the overhead and cost of a system that provides more functionality than required. And vendors can focus on niche markets with specialized systems that implement the requisite profiles.

It should be emphasized that the AEP approach supports portability on an intra-domain basis, and not on a global (inter-domain) basis. Since portability across domains is not supported, domains must be selected and defined with care. The issues of AEPs and portability are further discussed in Section 2.1.5.

Five profiles are being defined by POSIX's working groups: (1) supercomputing (P1003.10), (2) transaction processing (P1003.11), (3) real-time processing (P1003.13), (4) traditional interactive multiuser system (P1003.XX TIMS), and (5) multi-processing support (P1003.14). These working groups are attempting to define the needs of the specific application area, identify standards available to meet those needs, and close any gaps that exist between desired capabilities and standards. To assure that standards do not conflict but work together on a platform in a predictable way, the relationships between the standards is also being addressed. Following is a detailed description of the activity undergone in each of the existing profile groups [Emerging 1990, 19-20].

2.2.4.1.1 Supercomputing Profile P1003.10

Working group P1003.10 is defining a profile to support application and programmer portability in POSIX-based supercomputer environments. The profile developers have already identified existing standards applicable to P1003.10; they range from user interface standards to languages and networking standards. The following supercomputing functions still need standardization:

- Batch system administration and network definition
- Checkpoint recovery

- Resource manager
- Mass storage/archiving facilities
- Multiprocessing capabilities

Working group P1003.10 will continue on with their work and develop the necessary POSIX extensions to provide the missing supercomputing functionality.

2.2.4.1.2 Transaction Processing Profile P1003.11

Working group 1003.11 is defining a profile to support on-line transaction processing (OLTP) in POSIX environments. The profile will support OLTP for both a distributed environment and an environment in which the transactions are located on the same host. P1003.11 has already begun to identify already existing standards and OLTP standards that are under development. There are plans to work with the following standards groups:

- POSIX Remote Procedure Call (RPC)
- European Computer Manufactures Association (ECMA) 127
- POSIX 1003.4 Real-Time Extensions
- ISO Distributed Transaction Processing
- X/Open Transaction Processing (XTP)

After more work is done with the existing application standard groups, P1003.11 group will be able to recommend ways to obtain missing OLTP functionality [Emerging 1990].

2.2.4.1.3 Real-Time Processing Profile P1003.13

The prospect of a real-time POSIX standard raises the issue of what P1003.4 compliance means. If compliance requires all real-time vendors to implement the entire P1003.4 specification, a real-time system's responsiveness would be reduced and memory requirements for a space-limited system would increase. For this reason, P1003.14 group was formed to define several real-time profiles to be used by different real-time applications. These applications are currently defined as follows

- Low-end, embedded systems that require minimal functionality
- Mid-range, real-time systems that require medium level real-time requirements.
- High-end, software real-time systems that support full functionality of P1003.1 and P1003.4 [Emerging 1990]

2.2.4.1.4 Traditional Interactive Multiuser System Profile P1003.XX

The profile group for Traditional Interactive Multiuser Systems (TIMS) was recently formed to specify the profile of a typical UNIX environment.

2.2.4.1.5 Multi-Processing Support Profile P1003.14

The profile group for Multi-Processing Support was also just formed. Its purpose is to extend POSIX functionality to a multiple processor computer architecture.

2.2.4.1.6 Other Candidate Profiles

POSIX 1003.0 references six other profiles that are under consideration:

- Software Development Environment *
- Office Automation
- Autonomous Systems
- High Availability systems *
- Embedded Systems
- PC/Workstations *

2.2.4.2 Testing Conformance to POSIX

"Conformance testing involves testing both the capabilities and behavior of an implementation and checking what is observed against both the conformance requirements in the relevant POSIX standards and what the implementor states the implementation's capabilities are" [IEEE 1990, 8]. Traditionally, conformance testing has been done in a piecemeal manner for each standard; no assumptions existed about the options and parameters within a standard or the relationship that existed between standards. But now, with the advent of Application Environment Profiles, this ambiguity will be eliminated and more comprehensive testing environments will be provided.

The original intent of the POSIX 1003.1 working group was that conformance to the standard would be a condition claimed by operating system vendors and judged in the marketplace. But because of the complexity of the POSIX standards, it became clear that a test suite was needed to evaluate conformance. For this reason, the POSIX 1003.3 working group was formed in 1986 [UniForm 1990a] It was tasked with publishing a standard that contained (1) general requirements for how test suites should be written and administered and (2) a list of test assertions, showing exactly what had to be tested. In other words, P1003.3 does not define how to test, but what to test.

P1003.1 distinguishes between *operating system implementation* conformance and *application* conformance. These two aspects of conformance reflect the two sides of the interface defined in the standard: the operating system side and the application side. A conforming P1003.1 operating system implementation must include a certain set of minimum functionality. It may also include extensions if they don't conflict with the functionality specified in the standard. A conforming application on the other hand may not

* These profiles would be of greatest importance to WAM portability.

use any extensions and is prohibited from depending on any undefined, unspecified, or implementation specific behavior. It must be able to operate in the absence of any of the optional features of P1003.1 [IEEE 1988, 24-25].

POSIX defines three levels of conformance for applications that correspond roughly to the portability of an application: (1) strictly conforming, (2) conforming, and (3) conforming using extensions [IEEE 1988,24-25]. Strictly conforming applications are highly portable; whereas, the portability of conforming applications that use extensions is dependent on the extensions used. If the extensions are widely implemented in operating systems, it may not be difficult to port an application to a POSIX implementation. But if the standard extensions used are not implemented in a particular, POSIX-compliant machine, portability will not occur.

While IEEE does not certify conformance to a standard, other groups have taken P1003.3 and used it as the requirements specifications to write POSIX conformance test suites. NIST, X/Open, and AT&T have produced test suites for P1003.1 that meet the requirements of P1003.3 [IEEE 1990]. Once the other working groups define their standards and profiles, P1003.3 will develop conformance testing standards and test assertions.

2.3 APPLICATIONS PORTABILITY THROUGH ANSI SQL

This section of the document addresses portability guidelines for SQL. First, a brief description of SQL, as well as the current focus of the ANSI database standards group, is provided. Next, the different domains in which SQL is typically used is described. The issue of supersets and subsets and how they are addressed by ANSI SQL is then described.

2.3.1 Background

In 1986, the ANSI X3H2 Database Committee developed the first ANSI standard relational database language SQL [ANSI 1986]. ANSI SQL provided facilities for defining, manipulating, and controlling data in a relational database. The SQL standard has since been revised to incorporate support for integrity enhancement features and embedded language interface support [ANSI 1989].

The ANSI X3H2 committee currently expects to produce a new version of SQL, named SQL2, in the spring of 1992. SQL2 will provide support for schema manipulation, dynamic SQL, and many of the other "advanced" features found in today's commercial database management systems (DBMSs). The SQL2 specification will comprise three levels, the first being the current SQL, the second being about half of the new, improved, and easily implementable features, and the third specifying all of the new features. A

draft version of the SQL2 standard [ANSI BSR 199X] will be released for public review shortly, but was not available at the time this report was written.

2.3.2 SQL Application Domains

There are several different application domains for SQL. It is important to understand these application domains since the ease with which portable ANSI SQL applications can be developed is different for each domain.

A database administrator is primarily concerned with writing applications involving schema manipulation and storage management. Schema manipulation involves constructs such as CREATE, ALTER, DROP, and REVOKE for tables, views, and privileges. In addition, system tables (i.e., the database catalog) aid the database administrator in understanding the overall database. System tables allow queries which provide much information on all tables and columns existing in the database, such as who the owner is of these tables, when the tables were created, what users may update particular tables and views, etc.

ANSI SQL provides little support for the schema manipulation. Only CREATE constructs are defined for tables, views and privileges. However, complete schema manipulation constructs are usually provided in a commercial database since they are critical to database administration. This additional functionality is usually provided in a similar manner across commercial DBMS implementations.

ANSI SQL does not provide support for the concept of system tables. This functionality is also typically provided by commercial DBMS implementations, but the method of implementation is not common across implementation.

Application programmers often write code which accesses the database from a host language (e.g., COBOL or Ada), retrieves the data, processes it, and then perhaps produces a report. ANSI SQL provides two methods for accessing a database from a host language: the embedded calls approach and the module language approach.

In the embedded call method, the programmer codes (i.e., embeds) the database portions of his code in ANSI SQL within a host program such as COBOL or Ada. The embedded SQL is prefaced by a keyword so that a preprocessor may be used to scan for SQL calls and replace them with appropriate host language code which will then interface with the DBMS.

In the module language approach, the application programmer defines a host language subprogram which forms an abstraction of what is desired of the database call. Within the host language subprogram is a link to an object module which represents

compiled SQL statements. At execution time, the host program will execute the pre-compiled SQL statements, and then returns data to the host program.

An interactive user is concerned with direct invocation of SQL constructs such as SELECT, UPDATE, and INSERT. ANSI SQL provides no formal support within the standard for the interactive user. All interactions with the database are assumed either through the embedded or module interfaces. Most commercial database implementations offer an interactive SQL capability; however, there exist subtle but important differences between them.

The next version of ANSI SQL, SQL2, is anticipated to provide support for several different application domains by including additional schema manipulation facilities, the concept of system tables, and an interactive SQL facility.

2.3.2.1 ANSI SQL Supersets and Subsets

The ANSI SQL standard can be characterized as specifying the "floor" of the language, as it dictates the minimal syntax and semantics of SQL, and allows supersets (and, to some extent, subsets) to the language. As a contrast, for example, the Ada programming language standard seeks to define the "floor" as well as the "ceiling", by mandating no supersets or subsets.

There are about fifty "implementor defined" syntactical and semantical aspects published in the ANSI SQL standard. Most of these pertain to the implementation of numerics and precision issues. Others deal with issues the ANSI X3H2 committee deferred, such as character set specification, definition of the physical *newline* terminator, unusual states during a transaction, exceptional conditions, etc. It is unlikely that many of these issues will be clarified in the SQL2 standard.

2.4 COMPENDIUM OF PORTABILITY ISSUES

The concept of portability is intuitively appealing; users would like easily moving applications to new environments. Beyond saying that they want portability, however, program managers face the challenge of translating statements of want into contractual language, of implementing a strategy that enforces the contract, and of exercising sound judgement when faced with choices of conflicting wants or needs.

Achieving portability within the WAM Program raises several issues:

- What benefits will portability provide?
- Is portability consistent with other WAM goals?
- How can the requirement for portability be described?
- What means are available to promote portability?

- How can portability be verified?

These issues will be discussed in the following sections.

2.4.1 Benefits of Portability

Portability is an important characteristic for software in the WAM program because the WAM program strategy and the intended architecture are based on the ability to move applications software among different computers. Portability of software is generally believed to save money, improve the quality, and allow faster development of new capabilities. The expected savings in transferring a program from one computer to another follow when the cost of getting the program to work on the second computer (or environment) is less than the cost of redeveloping it and the cost of maintaining only one program (albeit in two versions) is less than the cost of maintaining two programs. The expected savings from reuse are similar except that a module is being recycled within several different applications.

Quality improvements are less obvious. One view of software holds that programs, when first written, contain a number of undiscovered errors. As the software is tested and used, these errors are discovered and corrected. If the corrections are made without introducing new errors, then the quality of a piece of software improves over time. According to this view, transporting software to a new computer or reusing modules in new applications is expected to result in software with fewer errors on the new computer or in the new application—hence better quality than newly written software.

When considering schedule improvements, portability and reusability seem to offer obvious advantages. In practice the evidence of benefits is more elusive, but the argument might be stated as follows: realistic planners do not expect that software can be transported or reused without some added effort; they know it is not free. However, the effort needed to transfer software to a new environment or to reuse a module is expected to be significantly less than the effort to produce the first item.

2.4.2 Architectural Considerations

As a separate effort under this tasking, we are providing a description of the target architecture for WAM and generic CCIS. The target architecture implements the concepts of layering functions so that changes within one layer do not affect activity within another layer. The architecture is outlined in [WWMCCS 1989, Encl (6)] and will be more fully described in a separate IDA report. This architecture will play an important role in promoting portability of applications software as shown in Figure 1. The interface between application software and its environment (i.e., operating system, data management, and other processes) will be clearly defined, using open standards wherever

possible. To the extent possible, applications are designed to be independent of particular features of their environment.

In reviewing available documentation concerning WAM, we learned that the short term plan for connecting the workstation (WIS workstation) to the WWMCCS ADP calls for using terminal emulation [WWMCCS 1989, 24-28]. In particular, the workstation is to emulate a VIP 7705W terminal and will connect with COBOL applications on the DPS 8 mainframe via the existing Datanet. We view this approach as an interim solution to user access to computing resources and not as a long term approach to providing connectivity in a heterogeneous, open systems environment.

Ada provides features that support the development of portable software. The ability to encapsulate machine dependent features allows programmers to limit the number of changes to be made when programs are moved to new computers. Using standard operating system interfaces and separating data from applications also promote portability. We believe these techniques should be used for any application, including terminal emulations. However, the benefits of portability will be limited as long as the architecture remains dependent on vendor unique operating systems and terminal protocols and the data remains bundled within particular applications.

We have participated[8] in the design and implementation of distributed systems based on both the terminal emulation paradigm and on other approaches such as interfaces based on shared access to data. Based on our experience, we believe that terminal emulation, albeit a quick way to achieve integration, has serious limitations if a program intends to comply with GOSIP or to implement automated data analysis tools.

2.4.3 Standards

The standards discussed in this paper are evolving to provide more complete and current frameworks for software systems. This evolution can be viewed as slow but steady progress toward open system architectures. Application systems developed during a particular phase in this evolutionary process will require some change and update later to remove non-standard extensions or workarounds when a more mature open system framework is available.

2.4.3.1 Ada

Although Ada was accepted as a DoD and ANSI standard in 1983, the standard allows some flexibility for compiler vendors to implement optional features (Chapter 13,

8. The portability guidelines presented in Appendix E were developed for one such effort. Another effort that involved the terminal emulation approach for its early phases was the Tiger Paw System, in use at the joint agency El Paso Intelligence Center (EPIC).

for example, describes implementation dependent features.) In addition, the DoD has approved specific interpretations on the standard. These interpretations, called Ada Issues [ACM SIGAda1989], as well as proposed interpretations by the ISO Ada URG, may also affect portability.

An effort is underway to develop an update to the Ada standard that will incorporate additional features as identified by users and accepted by the interested parties. This effort is called Ada 9X. We expect the revised standard to be available in 1993. The process of gathering requirements from the user community has been completed. Users will have their next opportunity to influence this standard when the draft versions are circulated for comment.

But all of this discussion is not to suggest that it is difficult to construct portable software using Ada or that the Ada standard is deficient. On the contrary, proper use of the Ada package feature to encapsulate machine-dependencies, the separation of program specifications from the implementation details, and user-definable typing will all assist in making Ada software portable. Nevertheless, following the Ada standard (or following a portability guide) does not, by itself, guarantee that the application will have a particular level of portability.

2.4.3.2 POSIX

The POSIX standardization effort is still young and under development; therefore, some changes to the POSIX family of standards should be expected to occur for several years. Customers cannot, therefore, simply demand that suppliers conform to POSIX. On the other hand, neither should the user simply accept any operating system that the supplier chooses to offer. Allowing suppliers to choose the operating system has historically resulted in users being "locked-in" to proprietary systems. We believe that since the operating systems portion of the POSIX Open System Environment (OSE)[9] is based on UNIX,[10] the WAM Program decision to use UNIX-based systems that conform to POSIX systems, and relevant ISO standards (e.g., language and data communication standards) is correct.

There are three potential problem areas to be considered when using a POSIX OSE. Namely, POSIX allows for (1) optional features, (2) multi-semantic features, and (3) extensions. Because the POSIX standards are designed for general application and

9. The POSIX family of standards forms the basis for open systems environment. Therefore, we will refer to a system based on the POSIX family of standards as a POSIX-based Open Systems Environment (POSIX OSE). The reader is referred to Section 2.2.3 for further information on open systems.

10. POSIX P1003.1 is based on tradeoffs between the AT&T System V Interface Definition (SVID) and the Berkeley System Definition (BSD) versions of UNIX.

not specific project needs, it is unrealistic to expect a perfect match between POSIX features and project requirements. For this reason, it is equally unrealistic to prohibit the use of optional features, multi-semantic features and extensions. Nevertheless, all three cases must be examined carefully since their use can affect the portability of applications developed using a POSIX OSE. An example of the variation allowed within a POSIX standard, in this case POSIX P1003.1, is given in Appendix B. In all cases, the use of these features should be scrutinized. In particular, justification for their use and proper isolation and documentation of modules that depend on these features must be employed.

Optional features and extensions present a portability problem since they may not be implemented by every vendor. In the case of extensions, the effect on portability is major since the extension and its semantics are not a part of the POSIX standard. As a result, the use of optional features and extensions should be isolated and documented so that the application can be easily maintained and ported. Further, the use of these features should be justified by a demonstration that either the functionality or required performance characteristics cannot be acquired without their use. When performance is the only reason for using options or extensions, a portable version of the code should be given in the documentation.

Multi-semantic features may be implemented by selecting one (or a subset) of the allowed behaviors for the feature. Thus, multi-semantics features will also create portability problems if the correct function of the system depends on a subset of the allowed behaviors of the multi-semantic features. Once again, the use of these features must be controlled by the WAM Program Office so that the applications minimize their dependence on particular behaviors. In the case of multi-semantic features, a different type of control is needed since the use of the feature itself is not the problem; rather, the expected behavior is the problem. Whenever one of these features is used, the system developer should provide documentation that states whether or not the use of the feature is dependent on any subset of the allowed behaviors. If the use of a feature is dependent on a subset of the allowed behaviors, a justification similar to that provided for optional and extended features must be provided.

By controlling the use of these three types of features (optional, multi-semantic, and extensions), the user is better able to acquire portable systems. In those cases where these features cannot be avoided, their use must be properly isolated, documented, and justified. These control techniques will reduce the cost of transporting the software and provide the Program Office with useful information on deficiencies with respect to the POSIX OSE. This information can then be easily relayed to the IEEE and other standards bodies for future improvements.

There are three avenues of participation in the POSIX standardization process: (1) the working groups, (2) the balloting groups, and (3) independent review.

The working groups are responsible for developing draft standards. To accomplish this objective, POSIX holds open meeting on a quarterly basis. All of the P1003 working groups, as well as other working groups (P1201, 1237, P1238) that fall under the POSIX umbrella, meet at the same site at the same time. A list of working groups and points of contact is given in Appendix A.

The balloting groups are responsible for reviewing and approving the draft standards prepared by the working groups. Participation in the balloting groups is open to any member of the IEEE or the IEEE Computer Society. However, it is conventional practice that members of the balloting groups also participate in working group meetings or independent review.

Independent reviews can be offered by any interested individual or organization by obtaining a copy of the current draft and providing comment in a form suitable for consideration by the working group.

All three methods of participation can have an effect on the quality and functionality of the POSIX standards. The goal behind participation is to influence the standards so that they will better reflect the needs of the WAM Program Office than is otherwise possible. To be effective in achieving this goal requires a clear understanding of needs, and this is not without costs. First, the WAM Program Office should document and justify the use of all optional features, multi-semantic features, and extensions as shown in Appendix B. This will provide a basis for determining the needs and focusing participation in the POSIX standardization process. Second, the WAM Program Office will have to commit resources towards the standardization process, namely, personnel to participate in the POSIX standardization process. Selecting the avenues of participation and the level of personnel support will depend on the disparity between the POSIX standards and the needs of the WAM Program Office. Points of contact are shown in Appendix A.

2.4.3.3 SQL

The ANSI committee that is developing the SQL standard expects to produce a new version of the language, SQL2, in 1992. SQL2 will provide features which are needed, but are not contained in the current version. A draft version of SQL2 is to be released soon. Because the present version does not contain all the features needed for database management, and those features are included in SQL2, users could encourage suppliers to comply with the draft standard. We do not believe that mandating use of

draft standards is a good policy. As a result, until SQL2 becomes final, SQL can provide some help in developing applications that have good portability with respect to the data management system, but implementation dependent features will remain. Customers who deal with these non-portable features by limiting themselves to the database mangement systems of a single vendor will be at risk of getting "locked-in" to proprietary solutions.

The ANSI review process that results in the acceptance of SQL2 and the subsequent NIST review that leads to acceptance of the standard as part of a FIPS offer opportunities for the WAM Program to ensure that the standard incorporates features needed by WAM.

2.4.4 Portability Requirements

Although portability[11] is a desirable attribute, it is not without attendant costs. The techniques that are most effective in enhancing the speed of execution or making the most efficient use of memory may be the same techniques that inhibit portability. If speed of execution is absolutely critical for an application or module, then portability may be less important. Similarly, some modules or applications may perform functions that are so narrowly defined as to make the likelihood of their reuse very low. A particular concern for WAM is that applications written now might, of necessity, interface to their environment by conforming to a specification that is not "open," but at sometime in the future those same applications might be expected to use an ISO standard. Finally, there are incremental costs associated with developing and testing software that has higher levels of portability.

The Mission Need Statement (MNS) for the Joint Operation Planning and Execution System (JOPES) [DoD JCS 1989] establishes the validated requirement for a major part of the WAM program. It identifies three alternatives for satisfying the requirement: developing a major new system, integrating and incrementally enhancing existing systems, and integrating new capabilities with existing systems. Each of the alternatives would benefit from having software with good portability, but the MNS does not establish any particular level for it.

The WWMCCS ADP Modernization (WAM) Decision Coordinating Paper (DCP) [DCA 1989, 10-4] identifies the Apple MacIntosh IIx plus four other computers with qualified capabilities. The other four computers are described in requirements contracts for 1) Air Force Desktop III and TEMPEST II contracts, 2) Air Force Standard Multiuser Small Computer Requirements Contract, 3) Navy Database Machine

11. This discussion about portability could be applied to reusability as well.

Contract, and 4) Navy Desktop Tactical Support Contract.

We believe that applications software will be moved to more than the five computers identified in the DCP. The computer manufacturers and software vendors have, for several decades, continued to introduce products that 1) perform the same functions faster, 2) perform new functions, 3) occupy less space, and 4) cost less than previous products. Moreover, the current trend is for producers to offer products that can be connected together according to an open systems model so that customers can mix products from different vendors. We see no reason to expect that these trends will be reversed.

We anticipate that CCIS users will wish to take advantage of more powerful hardware and that they will want to move their applications programs from older hardware to the latest models. The MNS defines functions that will be performed by WAM software applications. We believe these functions will continue to be required when, because of improved technology or competitive reasons, users replace the original WAM computer hardware with new equipment. Portability will prove to be a valuable feature of the applications when they are moved easily and quickly to the replacement computers.

To be effective, requirements must be translated into measurable, contractual terms (DoDD 5000.3 [DoD 1986], DoDD 5000.28 [DoD 1985], USAF R&M 2000 [USAF 1987, 31-33]). Establishing a particular, quantified goal for portability is a more difficult task than just determining that portability is desirable. It includes, first, defining measures for portability and, second, establishing desired levels for them.

The validity of metrics for software is a topic of some debate in the technical literature. A sampling of the discussion can be found in [Evangelist 1984, 534-541; Halstead 1977; McCabe 1976, 308-320; and Evangelist 1983, 231-243]. Without engaging in the metrics debate, we believe that portability must be defined in measurable terms if the WAM Program is to include it as a contractual requirement. Recalling that portability is defined as the ease of moving software, the first problem is to translate "ease" into a measurable attribute. The two traditional measures of ease (or lack thereof) are time and money associated with porting or reusing software. There are legitimate arguments for using each, either in absolute terms or in relative terms as a portion of the porting (reusing) effort compared to the original design effort. In fact, for a particular scenario, there will be functions that map one measure into the other.

The problem with selecting either time or money as a portability measure is that the actual value is determined by many factors that are independent of the application being moved. These factors include the compilers, operating systems, hardware, and databases on the old and new machines. Depending on the particular combinations involved, one application could be moved quickly at low cost or the move could be very

difficult. Another factor involves the agency performing the move. If an inexperienced or unqualified team attempts to move an application it could have a very difficult challenge accomplishing the same task that another team could do easily.

An ideal portability measure would be a useful predictor of the future cost of moving an application to a new environment, would be free of bias regarding the competence of the team performing the move, and would be easy to calculate. Finding an ideal portability metric beyond the scope of this task. We consider software metrics in general to be an open research topic. However, source lines of code (SLOC) is one simple metric that is easy to calculate and is (to a degree) independent of the skill of the transporting team. SLOC is used in many software development cost models (albeit not as the only factor) and there is evidence that SLOC is strongly correlated with the cost and time of developing software [Myers 1989, 92-99].

Having selected an appropriate measure, and having determined how to go about assigning a value for that measure to a particular software object, a program manager must determine what value of the measure to apply to each software object.

Portability may be more important for some software objects than for others. These objects are found by software engineers who analyze the system architecture to find those objects that would provide the greatest return on the investment needed to improve their portability (or reusability). We are not aware of a proven method for conducting such an analysis, but instead discuss in Sections 2.4.5 and 2.4.6 two alternative approaches for establishing particular levels of each characteristic.

2.4.5 Alternative A: A Program Strategy Approach

One approach to establishing some particular level of portability is to derive the desired levels as part of a program strategy. The software applications that are developed for WAM can be expected to outlive the hardware on which they are installed. This is true for the software developed initially for WWMCCS. In the case of WWMCCS, the lack of portability of the applications software is one reason that moving to non-proprietary software environments is very difficult.

The WAM strategy is to move to an open systems environment through incremental development. The major participants (Services, Unified and Specified Commands) will develop some of the software applications and others will be developed directly under the supervision of the WAM Program. Command centers that are part of the CCIS are expected to have hardware from a variety of suppliers. DCA is responsible for developing an infrastructure that allows the applications developed by different sponsors to be integrated into a common system. DCA is also defining an architecture that promotes

portability of applications among sites that conform to the architecture.

The particular level of portability that is appropriate for individual applications may depend on the nature of each application. For some applications, portability may be unimportant and for others it may be very important. Applications that are expected to be long-lived and widely used would probably need high portability.

The particular level of portability needed for an application should be traceable to some mission need. As noted in Section 2.4.4, portability is not explicitly mentioned in either the MNS or the TEMP. However, we believe that it is possible to create a traceable requirement for portability by considering a requirement to replace hardware on a regular basis.

If a plan for hardware replacement is made part of the overall program support plan, then a framework for determining portability requirements can be constructed. The idea is to anticipate the need to replace hardware, to expect to gain certain advantages by buying the replacement hardware from a marketplace of competing suppliers who provide products that conform to open system standards, and to compare those advantages to the expected costs of moving applications to the new hardware. We do not claim any particular level of accuracy for this approach, but we assert that it is a basis for determining required levels of portability and for conducting tradeoffs of portability against other characteristics such as performance or development cost. An example of an approach for determining portability requirements based on program strategy follows.

- Assume that CCIS equipment will be replaced on a five-year cycle.
- Assume that the price and performance of possible replacement equipment improve according to historically established trends.
- Estimate the expected savings associated with purchasing replacement hardware in five years.
- Estimate the amount of applications software that must be transferred from the current equipment to the replacement devices.
- Estimate the cost of transferring the existing applications to the new computers and environments, assuming various levels of portability.
- Estimate the cost of achieving various levels of portability as the applications are being developed.
- Translate costs and savings into a portability metric using an appropriate model.
- Establish minimum and desired levels of portability (using an appropriate portability metric) by comparing expected costs and savings.

The example is presented to illustrate the approach; the actual analysis is more complex than this.

2.4.6 Alternative B: Portability as a Quality Indicator

Tracing particular target values for portability measures to requirements is difficult, but it can be done if the problem is viewed from a different perspective. If we assume that greater levels of portability are achieved though application of discipline in the total process of software engineering and that discipline is an essential ingredient of a process which produces quality products, then we can conclude that portability is an indicator of quality. Although portability is not the only indicator of the quality of the software development effort, it is a useful one. This use is outlined below.

- Assume that portability is an indicator of the quality of the software engineering process.
- Establish a procedure for gathering data that can be converted to a measure of portability.
- Apply that procedure to a representative sample of software applications that are comparable to the software being developed for WAM.
- Examine the distribution of the calculated portability levels.
- Establish a minimum acceptable level of portability based on the sample data and make this level one of the quality indicators.
- Establish levels of portability that can become part of the criteria for incentive award, if incentives are part of the procurement strategy.

Without some procedure for measuring the portability and reusability of delivered software and some indication of what levels are expected, there is, in our opinion, little likelihood that either attribute will receive the attention needed to produce significant benefits for the WAM program.

2.4.7 Communicating Portability Requirements

After establishing the desired goals for portability, the WAM program must find a way to communicate them to the contractors who will actually design and develop the system. The communication method will be different for the various stages of the acquisition process. For example, before a contract is awarded, the program office should state its portability requirements as clearly as possible and evaluate proposals for (among other things) the credibility of the contractor's understanding of the issue and planned approach.

2.4.8 Evaluating Portability

After a program has decided how to quantify portability, has established a requirement to achieve some level of portability on a particular application, has included that requirement in contractual terms, and has established management controls to ensure that the portability requirement is satisfied, the final step is determining whether delivered software satisfies the contract.

Once a contract is awarded, the development process defined in DoD-Std-2167A (as tailored) begins. During this stage, the contractor will develop preliminary and detailed designs for successive software releases. The program office will be concerned with ensuring that the design satisfies requirements and that cost and schedule goals are met. By preliminary design review the contractor's plan for using portability guidelines can be evaluated, and by critical design review the design should clearly show where the software is dependent on particular elements of the target environment.

After the software is written it can be inspected to find constructs that are difficult to move, or constructs that violate particular guidelines. The inspection does not guarantee that software will have a particular level of portability, but it can identify software that will have poor portability characteristics. The advantage of inspection is that it can be done by an automated tool, is cheap, and can be accomplished quickly.[12]

Although inspection is the fastest method for evaluating portability, it is not necessarily the most accurate. A customer discovers the portability of software when the software is moved to a different computer. However, the contractor cannot wait for an indeterminate period until the software is moved to a new environment before receiving final payment for work accomplished. Because the real move might happen years after the software is delivered, we believe that a test is needed to satisfy the customer that portability goals have been met. Once software is accepted as satisfying the application requirement on the first target system, the customer can apply the test to find out exactly how portable it is. If the test procedure and criteria have been described[13] in a Test and Evaluation Master Plan (TEMP) and in more detailed documentation that is produced during the development, then these tests could become part of the acceptance testing of the software. Moreover, if the test procedure is known to the developer and the developer is aware that passing the test is part of the acceptance process, then the developer

12. Two tools that evaluate Ada portability are Ada Metrics Analysis Tool (ADAMAT) and the Standards Checker. ADAMAT is a product of Dynamics Research Corporation and the Standards Checker was developed as part of the Common Ada Foundations toolset by Naval Ocean Systems Command as part of an earlier WWMCCS program.

13. The WWMCCS ADP Modernization (WAM) Test and Evaluation Master Plan (TEMP), 20 April 1990 contains requirements for mission utility, responsiveness, availability, interoperability, reliability, security, operations in a degraded mode, maintainability, and usability. It does not address portability.

will have a clear incentive to implement procedures that promote portability.

The problem of deciding which computers should be used to test portability is difficult. First, the customer should decide which classes of computers are reasonable candidates for hosting the application. By class of computer, we mean broad categories such as personal computer, desk-top workstation, minicomputer, mainframe, supercomputer, and special purpose machine (e.g., database machine). Within a class of computer, there might be differences in word length, byte order, instruction set, memory architecture, registers, or even arithmetic algorithms. Software environments on a computer can vary if different compilers, operating systems, or database management systems are used. Demonstrations that show variation in as many of these factors as possible are more robust predictors of the actual future portability of the software, than those that vary in only a few factors.

2.5 PORTABILITY RECOMMENDATIONS AND GUIDELINES

The WAM Program is committed to achieving an open system [DCP 89, Appendices G and R]]. We believe that the decision to select an architecture that implements an open system will ensure the greatest benefit from portability. Such an architecture is outlined in [WWMCCS 1989, Encl(6)] and will be described in a separate IDA report. Moreover, portability is just one of the characteristics of software that should be considered during design. In some cases developers might have to trade-off portability to achieve faster execution speed or to comply with some memory restriction. The recommendations and guidelines are offered with the assumption that they will be applied within the context of developing a system using generally accepted software engineering principles.

Good application software portability is the result of thorough planning and execution during the software development effort. Developing a plan to achieve required portability is an important step for the WAM Program. The most important steps in developing a plan are outlined below and presented in greater detail in Sections 2.5.1 through 2.5.6.

- Quantify what is meant by portability.
- Decide how much portability is needed for each application and include this information in the statement of requirement.
- Identify standards and practices that promote the development of software that satisfies the requirement and, depending on the type of contract being used, either encourage or coerce the contractor to follow those practices.
- Ensure that policies are being carried out during development through government reviews.

- Test the delivered software to ascertain that it has the required characteristics.

2.5.1 Quantify Portability

> **Recommendation 1** Use source lines of code (SLOC) as a basis for measuring portability. Measure the number of SLOC that differ between the application that executes in different environments. Define a function to convert this measure to a portability value.

Software portability is defined as a characteristic of software; it is not something that software either has or does not have. We believe that if portability is to be achieved, then the concept of portability should be defined in terms that can be easily and accurately measured.[14] The IEEE definition of portability does not provide a basis for a sound program because it uses the relative term "ease" and the definition provides no guidance on how to measure "ease." In addition to the traditional measures of "ease," cost and time, we can add "source lines of code." In the case of portability, any of these could be used, either singly or in combination. They could also be used either as absolute values or in normalized form (as ratios). The individual measure to be used on WAM will depend on the needs of the program as well as the ability of the program office to gather the data.

Because SLOC is already used for many management models, because it is easily measured, and because it has strong correlations to time and cost, we recommend that SLOC be used as a basis for calculating portability.

The exact procedure for using SLOC will depend on the application under consideration. We expect that different methods would be used for a very small applications and large applications. The procedure might include the following steps.

- Measure the number of SLOC that differ between the application that executes in different environments.
- Define a function to convert this measure to a portability value.

The function used to convert measure to a portability value may differ according to the particular application and the software development model used. The simplest function would be the count of SLOC that differ (i.e., lines that were added, deleted, or changed when the application was moved). A slightly more advanced function is the ratio of changed SLOC to the size of the application on the original host. The selection of individual functions will be determined by a detailed analysis of each application.

---

14. The term "operational definition" is used by Deming [Deming 1986] to denote definitions that can be subjected to independent testing. He says, for example, that requiring that a blanket be 50 percent wool can lead to very different interpretations because the testing method is not specified.

2.5.2 Establish the Requirement

Recommendation 2 Develop a model of equipment replacement factors as suggested in Section 2.4.5 and use it to establish quantitative portability requirements for each WAM application.

The user's needs define the applications and the applications themselves imply the feasibility of being moved. Applications that satisfy valid, traditional military needs and that are expected to be needed in the future are going to be moved to new computers or environments. These applications need high portability. Exactly how much portability is needed and affordable is less obvious.

In Sections 2.4.5 and 2.4.6 we outlined two approaches that could lead to the establishment of quantitative goals for portability. The first approach is more difficult, because it involves creating models of the costs and benefits of future equipment purchases and software installations. However, once done it gives the CINCs and Services a tool for planning and budgeting. We believe it is the better approach.

2.5.3 Identify Supporting Practices

Recommendation 3 Use standards to promote the development of portable software.

We believe that standards are available for the programming language, the operating system, and the database interface. Although the standards are subject to change and do not, by themselves, guarantee that applications will be portable, they are a necessary part of the process. Specific guides for using standards to promote portability within the applications programs, at their interfaces to the operating system, and at their interface to the database are given below.

Ada

Unless specific waivers are approved, require applications programs to conform to ANSI/MIL-STD-1815A-1983. It specifies the form and meaning of programs written in Ada. Its purpose is to promote the portability of Ada programs [ANSI 1983, 1-1].

POSIX

Require operating systems to conform to IEEE Std 1003.1-1988 for interfaces to system services such as process management, signals, time services, file management, pipes, file I/O, and terminal device management.

- While POSIX standards are being completed, mandate that contractors use

UNIX-based systems or conforming POSIX systems. The POSIX operating system interface standard (P1003.1) is based on tradeoffs between the AT&T System V Interface Definition and the Berkeley versions of UNIX; the final version of P1003.1 is expected to be closely related to UNIX.

- Discourage use of POSIX AEPs. Care must be taken in selecting a POSIX AEP for domain specific applications. A POSIX AEP is a subset of POSIX OSE, plus an arbitrary collection of options, parameters and extensions. As a result, the use of POSIX AEPs will restrict the portability of application to those environments (domains) that are supported by the POSIX AEP. This must be considered if and when POSIX AEPs are selected.
- Allow use of optional features, when such use is justified and documented. The use of optional features, multi-semantic features, and extensions to the POSIX standards should be scrutinized. In particular, justification for their use and proper isolation and documentation of modules that depends on these features must be employed by the application developers. All of the POSIX standards, including the AEPs, are general in nature. Thus, it is unrealistic to prohibit the use of these features. Nevertheless, all three cases must be controlled carefully since their use can adversely affect the portability of applications. These control techniques will not only reduce the cost of transporting the software but will also provide the WAM Program Office with useful information that can then be relayed to the IEEE and other standards bodies for future improvements.

SQL

Require conformance to FIPS PUB 127-1 for all procured SQL implementations.

- Discourage the use of nonstandard SQL language features. ANSI SQL provides a basic capability to access a relational DBMS. There exist many additional features in current commercial SQL implementations. Some of these features are critical to many SQL users: schema manipulation language, date/time data types, support for interactive queries. Other features, while not critical, are extremely useful and may not be implementable within the current ANSI SQL standard.

FIPS PUB 127-1 aptly describes the issue of nonstandard SQL usage with the following paragraph: "Nonstandard language features should be used only when the needed operation or function cannot reasonably be implemented with the standard features alone. A needed language feature not provided by the FIPS database languages should, to the extent possible, be acquired as part of an otherwise FIPS conforming database management system. Although nonstandard language features can be very useful, it should be recognized that their use may make the interchange of programs and future

conversion to a revised standard or replacement database management system more difficult and costly" [FIPS 1990, 3]. Appendix C of this document identifies all "implementor-defined" aspects of ANSI SQL language. Annotations are provided to SQL application writers regarding the use of implementor-defined SQL constructs.

> **Recommendation 4** Ensure that WAM program interests and requirements are addressed in the standards definition processes and participate in the balloting on proposed standards. Continue to stay informed about new interpretations and implementations of approved standards.

Each of the standards continues to evolve in response to new technology and changing users needs. We believe that programs such as WAM should be prepared for changes to the standards and should be prepared to take advantage of them. This is best accomplished by participating in the standards definition and balloting processes.

The following guidelines are provided to assist the WAM Program in ensuring that the standardization efforts meet the program needs.

- Participate in the Ada 9X effort. The Ada 9X effort to revise the current Ada standard is open to input from the government through its Government Advisory Panel. In addition, the proposed changes to the standard will undergo public and standards organizations reviews before it is finalized in 1993.
- Monitor Ada Issues and URG Issues.
- Participate in the POSIX effort. Monitor and participate in the POSIX OSE and AEP development and standardization efforts. Monitoring these efforts will keep the Program Office informed as to the direction of the standards. Thus, the Program Office will be better prepared to set policy and procedure for contractors and better prepared to make comments to the IEEE and other standards bodies during the public review process. Involvement will be especially important during the NIST public review for FIPS.[15] Participating in the efforts will allow the Program Office to influence the design of these standards to ensure that they address the needs of WAM before they are approved by the standards bodies.
- Participate in the NIST public review of SQL2 that accompanies its approval as a FIPS.

15. NIST is heavily involved in POSIX OSE and it is expected that NIST will develop FIPS and validation procedures for POSIX OSE and AEPs. Since FIPS are important to the procurement process, WAM's involvement in POSIX standardization will indirectly influence the resulting FIPS.

2.5.4 Monitor the Development Process for Compliance

Recommendation 5 Require contractors to describe their process (including use of portability guides) for producing software with the required portability and monitor the process to ensure that it is producing the desired results.

We believe there is a growing consensus that significant improvements in software products will only happen when the software development process is improved. Standards have an important role in promoting portability, particularly regarding the programming language, the operating system interface, and the database interface, but standards alone do not generate portable software. Portable software is generated by a software engineering process.

In some, but not all, cases the customer has a direct interest in a contractor's development process. When the customer is directly paying for the process, such as when the software is developed under a DoD contract, then the customer has an interest in ensuring that the best procedures are followed to produce software products. When buying a COTS product, however, the customer need not be concerned with the developer's process the quality of the product is the main concern. We believe that applications written for WAM in Ada and database queries fall under the classification that merits program oversight.

We believe that contractors are better qualified to define their own processes than are their customers, but we believe that customers are entitled to a evaluate contractor's proposed processes during source selection and that customers are exercising prudent concern when they ask for evidence that contractor's processes are under control during customer funded development.

The principal vehicle for implementing a software development process that improves portability is the portability guide (sometimes included in a style guide or standards and procedures manual.) Therefore we believe that the WAM program office should monitor the contractors' use of portability guides.

However, porting Ada software from one platform to another requires more than just guidelines. Because guides are only one of the tools that enhance portability and because there are several excellent guides available, we do not believe that the WAM program needs its own guide. An understanding of the issues and the options available to address each issue, plus developing and executing a comprehensive plan for implementing portable software are needed as well. The following guidelines are provided to assist in the development/procurement of portable applications software in Ada:

- Require each bidder on software development to demonstrate a comprehensive portability approach. Proposal evaluation should include consideration of the bidder's understanding of the issues associated with Ada and the likely effectiveness of their approach in addressing these issues.
- Require that bidders on software development identify the portability standards and guidelines[16] they will use. Such standards and guidelines do exist, but the bidder may propose to develop its own. We believe it is important for the bidder to justify its selection by providing the criteria used for selection and the procedures that will be followed in enforcing the guidelines. We believe that imposing guidelines on the contractors would be a mistake.
- Encourage contractors to use automated tools to confirm that guidelines are being followed.

    — Require the use of a tool[17] to validate that Ada code conforms to guidelines.

    — Require the use of the FIPS Flagger. The FIPS Flagger is a tool that can assist application programmers in developing portable application programs. FIPS PUB 127-1 requires a vendor to develop a tool which effects a static check of an SQL program and flag code that violates format or syntax rules or code that extends ANSI SQL [FIPS 1990, 4]. This Flagger can assist application developers in identifying nonstandard, potentially non-portable, SQL constructs.

2.5.5 Test the Result

Recommendation 6 Require software to pass an acceptance test which measures portability for applications that have a portability requirement.

The final step to ensuring that software applications have the required portability is to develop and conduct a testing procedure. Such a procedure will ideally provide assurance that the delivered software has the required level of portability and that it can be move to new computers or environments when the need arises. Some guidelines for

16. A review of several guidelines is included in Appendix D. From this review, we conclude that developers could equally select either *Portability and Style in Ada* by Nissen and Wallis, one of the SofTech guides, the Martin Marietta *Software Engineering Guidelines for Portability and Reusability*, or the Software Productivity Consortium's (SPC) *Ada Quality and Style - Guidelines for Professional Programmers*. Additionally, a previous IDA report that includes some comments on guidelines for portability is provided as Appendix E.

17. Several such tools are available, such as ADAMAT and the Standards Checker, but we are unable to make a specific recommendation. Tools are useful as a check that guidelines are being followed so the particular tool to be applied will depend on the guidelines being used.

developing the test procedure are provided below.

- The test should demonstrate portability across each factor that can be anticipated for the individual application. Factors that could change include the class of computer (mainframe, workstation, etc.), word size, byte order, definition of arithmetic, instruction set, memory hierarchy, bus structure, compiler, operating system, display, and database management system. We believe that more challenging tests have more value to the customer and therefore that tests should require software to be moved to sample computers from each class that is appropriate for the individual application.
- An IV&V agent should test the portability claims for delivered software. This will provide a greater assurance of compliance with software portability requirements.

3. REUSABILITY

3.1 INTRODUCTION

The reuse of software offers the potential of increasing productivity in building parts of the system and increasing the quality of the system. These increases in productivity and quality can be expected to result in cost savings, reduced development time, higher system reliability, and other benefits.

Although the authors of this paper acknowledge that major technical problems must be solved before the full potential of software reuse can be realized, we feel that some benefits can be obtained now. However, software reuse is difficult because of such non-technical factors as organizational structures, financial disincentives, and lack of specific contractual mechanisms that allow and encourage reusable software.

3.2 POTENTIAL BENEFITS OF SOFTWARE REUSE

A wide variety of benefits have been claimed for software reuse, including the following:

- Increased productivity
- Increased reliability
- Higher quality design and code
- Decreased development time
- Reduced need for testing
- Cost savings
- Reduced maintenance
- Lower risk
- An ability to build larger, more complex systems

These claims can be difficult to sort out because they are interrelated. In fact, most of these claimed benefits result from two principal factors: (1) increased productivity (of designers and implementors) and (2) increased quality (of design, code, and documentation). Increased productivity of personnel is perhaps the most common claim, and one that, if true, can legitimately result in cost savings, decreased development time, and an increased ability to build larger, more complex systems. Higher quality of code, design, and documentation generally can result in a reduced need for testing, reduced

maintenance, higher reliability, lower risk, and an increased ability to build larger, more complex systems.

These claims should be viewed with skepticism. There is little clear evidence that any or all of these benefits can be predictably achieved for any given software project. Much of the available data is for software developed under conditions that may not be easily replicated elsewhere, for specific applications that may not be very similar to WAM applications, or that involve small-scale tests that may not be very applicable to major industry software efforts. Data from many industry efforts is not available or is incomplete for competitive reasons. For any given software project, attempts to reuse code in an inappropriate way may well have a negative effect.

3.3 SOFTWARE REUSE BACKGROUND

3.3.1 What is Software Reuse?

Software reuse is not a new idea. In one form or another, it has been practiced almost since the beginning of computing, by reusing subroutine libraries.

The modern move to reuse has frequently been attributed to a challenge posed in an address by M. D. McIlroy at a 1968 NATO Software Engineering meeting in which he pointed to the success of the hardware components industry (which was beginning to produce standard components in the form of digital electronic chips) [McIlroy 1969]. McIlroy asked why an analogous industry could not be developed around software components. The extent to which this analogy is appropriate has been the subject of considerable debate, as has been the reason why, more than two decades later, such an industry has not developed.

One of the difficulties in understanding reuse is a result of the many definitions, and the lack of agreement among the experts. These definitions typically reflect differences in what contexts are considered legitimate and in what level of representation (from abstract knowledge to specific program code) is appropriate for reuse.

For example, the following definition takes a broad interpretation of the allowable context for reuse:

> Software reuse is the reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of that other system [Biggerstaff 1989, xv].

Another definition, however, strictly limits the context:

> Software reuse, to me, is the process of reusing software that was designed

to be reused. Software reuse is distinct from software salvaging, that is reusing software that was not designed to be reused. Furthermore, software reuse is distinct from carrying/over code, that is reusing code from one version of an application to another [Tracz 1989, page 36].

From these definitions, the following conclusions can be drawn:

- The present interest in and emphasis on reuse is a result not of any specific technical achievement, but instead simply reflects a way of thinking that accepts reuse as potentially important and as feasible. This view is moving toward considering all entities involved in developing a product, whether they are specifications, requirements, source code, object code, or documentation, as potentially reusable objects. Also, most experts assume that, in most cases, successful reuse of a component will require modification of that component.
- Experts have been concerned with whether truly reusable software must be specifically designed for reuse; some view the practice of unplanned reuse "software salvaging" as having much less potential payoff. [Tracz 1989, 36].

3.3.2 Software Reuse in the Development Cycle

Until recently, software reuse has occurred in a relatively unplanned, random fashion. Despite the debate over just what practices might be considered legitimate reuse, we believe that the greater the extent to which reuse is planned, rather than accidental, the greater the benefits that will be achieved and the lower the risk.

Typically, a more sophisticated attempt to apply software reuse involves four activities: domain analysis, component and tool construction, system construction, and development of a reuse library. The order that these activities are carried out can vary considerably depending upon the situation, as is discussed in the following paragraphs.

Domain analysis involves the detailed study of a particular application problem and how solutions to it can be developed for implementation with a computer. Most domain analysis today is done relatively informally by consultation with experts in narrow and relatively well understood domains. However, it is widely agreed that the broader and more detailed the domain analysis, the easier it will be to find commonalities among different problems that allow the construction of more general software components that can be reused. Thus, it is usually recommended that a broad, extensive domain analysis be done at the beginning of a software development effort. The usual barrier to this is that an extensive domain analysis is very costly, and an initial investment that is often hard to justify.

Component and tool construction involves the design and implementation of software components and whatever supporting tools are required to aid in their development.

The term component is frequently interpreted broadly and can refer to high-level software designs and test plans as well as source code. There is some overlap between domain analysis and component implementation, in that implementation issues must be considered in performing an effective domain analysis.

System construction involves the development of a system, using components that are newly designed with a system in mind (but also intended for reuse elsewhere) as well as components that may have been used elsewhere and that may well need modification.

The development of a reuse library involves the creation of a central source of information about what software components are available, their capabilities, restrictions on their use, and how they can be obtained. Such a library, also known as a repository, has some way of aiding search for a desired component. This is generally done with relatively simple techniques, such as keyword matching. More sophisticated systems might make use of a taxonomy or categorization scheme developed for a particular broad application area as the result of a domain analysis. A library may contain the actual software components and their associated documentation for retrieval by the user, or it may simply provide information about where they can be obtained.

How these different activities relate to each other depends on the nature of the system being built, the extent to which reuse is planned, and whether reuse is done primarily in a small group of developers or is attempted across a large project involving many different contractors. A small effort carried out within a single group can effectively reuse software even with a relatively informal domain analysis. Effective reuse in a large, complex, multi-organization project, however, has not yet been demonstrated, and attempting it will require an increased formality of domain analysis, a more organized way of planning component, tool, and system construction, and a reuse library.

3.3.3 Nontechnical Factors Enhancing or Inhibiting Reuse

The extent to which software reuse can be effectively practiced depends upon a number of organizational and economic factors, in addition to technical limitations. The following are some of these factors:

- Organization. Because practices and technology for reuse are still rather primitive, relatively informal techniques are used. These techniques can work well if they are systematically applied within a small group of software engineers who agree to use a common style, and can communicate with each other. Reuse is much less likely to be effective if software from one organization is attempted to be reused by another.
- Management Commitment. Because the design of reusable code requires

planning and up-front investment, and can in the short term reduce programmer productivity, it is most effective when there is a strong commitment by management to reuse, including incentives to individual programmers to reuse software. Approaches should be developed as part of the WAM contract that can help motivate contractors to have this commitment.

- Economic Disincentives. DoD contracting practices, particularly those concerning intellectual property rights for computer software, have been widely criticized as inhibiting reuse. One problem is that when software is delivered to the government under a software development contract, the government demands (under federal acquisition regulations) unlimited rights to that software. This is often perceived as excessive by contractors, and can prevent a contractor from recovering its investment in developing a library of reusable components. If a component is modified using government funds, the government may gain unlimited rights to the component. This can inhibit the contractor from delivering another system that includes a different version of that component to the government. There is much uncertainty about specific interpretations of these regulations, which is itself an inhibitor of reuse.
- Contracting Based on Appropriate Requirements. Reuse can also be inhibited if contracts and specifications are not written carefully that define appropriate requirements. Just as requiring that a certain level of portability is meaningless, as was discussed earlier in this document, it is also meaningless to require a fixed level of reusable modules, or that a certain amount of code in a system be reused code. Appropriate requirements can best be developed by means of an analysis of how reuse can be best accomplished in the WAM system, with a concept of operations document developed to serve as a basis for requirements for reuse.

3.3.4 Research Issues and Activities

Although many benefits of software reuse are achievable now, its full potential will only be realized when the technology for reuse is developed more fully by research.

We see seven significant technical issues that are being (or should be) investigated, with progress in any of the areas likely to result in enhanced reuse capabilities:

- Improved methods for domain analysis.
- Improved indexing and retrieval systems for reuse libraries.
- Improved conceptual understanding and representations for reuse.
- Methods for raising the assurance that software performs as expected.
- Reuse methods that take into account the fact that software not only carries out

a functional task but does so with certain resource utilization characteristics.

- Techniques for managing the increased number of parameters that are required for large components.
- Improved software tools for reuse.

The extent to which progress has been made in these areas, as well as progress in putting reuse into actual practice, will be major factors determining the extent to which reuse should be attempted in the WAM program. Progress should be monitored by, for example, sponsoring annual workshops in software reuse that can help determine the state of research and the state of practice.

Several DoD programs can be expected to develop technologies, standards, methodologies, and contractual and management strategies for software reuse. The STARS program, or Software Technology for Adaptable Reliable Systems, was established to improve software in DoD programs. Key goals are improving quality, reliability, and productivity, with software reuse a means to this end. Toward this end, STARS has funded the prototyping (by Boeing, IBM, and Unisys) of several software reuse prototype libraries, and other reuse technologies are expected to developed.

The Joint Integrated Avionics Working Group (JIAWG) was established by the Services in response to a mandate by Congress to identify common aspects of avionics equipment in three aircraft programs - the Navy Tactical Aircraft (ATA, currently known as the A-12), the Air Force Advanced Tactical Fighter (ATF), and the Army Lightweight Helicopter (LH, previously known as the LHX). As part of the JIAWG effort, a software reuse subcommittee has been developing plans and standards to support and encourage software reuse within each of these programs as well as across the programs.

The Strategic Defense Initiative is also expected to initiate efforts in software reuse, and efforts are under way to develop a reuse library toward this end at the National Test Bed in Colorado Springs, as well as work by GE Aerospace in Blue Bell, Pennsylvania, and work toward defining a conceptual model for reuse and in investigating contractual and legal issues at the Institute for Defense Analyses.

3.3.4.1 Domain Analysis

Methods that can help identify commonalities in related application problems that can lead to reusable components are badly needed. These methods include better theoretical approaches to understanding problem-solving, and software tools that aid in domain analysis. It is likely that research in expert systems and artificial intelligence in the areas of knowledge acquisition and knowledge elicitation, in automated knowledge acquisition tools, and in cognitive science approaches to understanding the mechanisms

of human problem-solving can help in those areas where expert human problem-solving is to be encoded into software.

There are several efforts under way aimed at improving domain analysis. Biggerstaff has been investigating what information is best captured during a domain analysis and what methods are best at capturing it. He has built several tools to prototype different approaches to domain analysis [Biggerstaff 1989]. In addition, the Software Engineering Institute has used the Common Ada Missile Packages (CAMP) products to investigate domain analysis [McNicholl 1988]. SofTech is also including domain analysis capabilities into their Reusable Ada Packages for Information Systems Development (RAPID) Ada component library.

3.3.4.2 Indexing And Retrieval Systems

In large-scale software reuse, especially across different organizations, the ability to find components that might be useful is critical. Current systems make use of approaches borrowed from the indexing and retrieval of textual documents, including keyword search strategies and classification taxonomies.

Software classification schemes provide a basis for a library catalog that will allow users to locate information they need. In fact, several classification schemes providing cross-indexes to available reusable objects may be essential to find and evaluate reusable software components. Proposed schemes include functional classifications, code taxonomies, and type hierarchies. None of these schemes, however, has been tested in prototype catalogs for full-scale libraries.

Two types of automated search techniques are immediately available for libraries: database query systems and literature search systems. Database query systems respond to specific questions that have concrete answers derived from stored data. Literature search systems, often based only on keywords in abstracts, provide approximate searches that may not find all references on a topic and sometimes return unrelated references. Approximate searches are useful when there is not enough information to formulate precise queries. Natural language query systems have been demonstrated for narrowly defined application areas and should be considered for future repository access.

Other approaches that are becoming popular include hypertext browsing and searching systems [Latour 1988], and object-oriented hierarchies. Proponents of object-oriented hierarchies tend to support a different philosophy in which reuse occurs as a natural result of dynamic interaction with an object-oriented system, rather than as a result of searching a separate library system for a specific desired component.

3.3.4.3 Conceptual Understanding And Representation

The problem of breaking a problem down into components and how components are best represented is important to developing reuse technology.

There is controversy about what entities or levels of representation are the most appropriate for definition as a component. Biggerstaff and Richter, for example, have argued that designs rather than implementations are the best level of abstraction for reuse [Biggerstaff 1987]. This is because implementations involve details that are specific to the application problem or target machine, limit the generality of the component, and inhibit its reuse. However, Booch [1987] has argued that to make reuse of components truly attractive, implementation issues must be addressed. In addition, Edwards and Baldo [1990] have argued that the implementation level must be considered because the major savings resulting from reuse occur during maintenance and reuse of designs alone will not provide the desired savings during the maintenance phase.

One approach to understanding how software can be divided into components has been to develop a formal model for software reuse. This approach was taken at the Reuse in Practice workshop [Baldo 1990], where the beginnings of such a model, the 3C Model, were produced. This model is an attempt to describe reusable software components based on three aspects: (1) the concept, (2) the content, and (3) the context. The concept refers to a description, at an abstract level, of what the component actually does. The content refers to the details of how the abstract concept is actually implemented. There can be more than one content for a concept, since generally there are many algorithms for implementing a given task, usually with different performance and resource characteristics. The context of the component refers to those aspects of the software environment that can give additional meaning to either the concept or the content. In the case of the concept, the context of a particular data type for an operation defined by the concept can, for example, provide additional meaning to that concept. The content also has a context which can alter the meaning of its implementation. (For a more detailed description of the 3C model see [Edwards 1990]).

3.3.4.4 Methods for Assuring that Software Performs as Expected

One frequently stated inhibitor of reuse is that programmers do not trust software that they did not write. Unless a programmer has confidence in a component from a library, he or she will not use it. Methods that can help assure that a component performs as expected can help address this problem (DeMillo 1988).

3.3.4.5 Resource Utilization Characteristics

Currently, reusable software components are both designed and classified based on a *functional* model, i.e., *what* the component does. Component selection, on the other hand, is driven not only by the function a component performs, but also its resource utilization characteristics. Resource utilization is a term that refers to the fact that each component requires various abstract resources from its environment in order to run (for example, memory space, processor execution time, file handles, access to a terminal, etc.). Furthermore, the types and amounts of resources are different for each implementation of each component.

Unfortunately, the resource utilization of a component is currently treated in an random manner. For effective reuse, it is important not only to have a model of the functional behavior of a component, but also a well-defined model (or set of models) of the resource behavior as well. However, no such model(s) are now in use. Instead, components are currently stored in a library along with simple performance information based on benchmarks measured on a specific target, and which only address memory or cpu utilization.

Without an effective and comprehensive means of capturing and presenting resource utilization information to a potential component user, it will be very difficult for resource trade-offs to be made during component selection. This will also make reusable components less viable for real-time or embedded systems, where resource performance is a critical concern.

3.3.4.6 Management of Parameters

It is frequently stated that the most savings from reuse results from reapplying relatively large components [Biggerstaff 1987]. As components become large, however, the size and complexity of the parameters that are required to drive them can result in a secondary requirement for parameter configuration assistance. The principal problem is the empirically observed limitation in the ability of humans to deal effectively with long lists of relatively unstructured parameters. Another more temporary problem is the inability of some compilers, particularly Ada compilers, to handle components with large numbers of parameters.

The tendency of large components to have many parameters results from the fact that large components are typically constructed from smaller components, which may each be constructed from still smaller components, forming a hierarchy. While many of these parameters can be fixed, large components may still include many of the parameters of each smaller component that it uses. Components at the highest level can thus

develop very large sets of parameters.

Approaches need to be developed to help humans deal with such large sets of parameters, and there has been work toward solving this problem, such as operating system configuration hooks, window system defaults editors, shells for selection and composition of components, shells for statistically well-designed experimental parameter search, etc.

3.3.4.7 Software Tools for Reuse

Once techniques in each of the previous six areas have been uncovered by research, production-quality software tools must be developed. Such tools are the key to moving these techniques from the research domain into practice. Without automated support, reuse techniques will only be slowly adopted, and will also be slow to show savings.

3.3.5 Recommendations

Software reuse has a significant potential for increasing productivity and quality, which can result in reduced cost, reduced development time, increased system reliability, reduced maintenance, and other benefits. However, software reuse is not yet a mature technology, and its potential is severely limited by the relatively primitive methods and tools presently available for software reuse, and by organizational and policy problems that can inhibit reuse. We therefore make the following recommendations:

- **Software reuse has a significant potential for the future, but is not yet a mature practice.** While some benefits can be achieved on a small scale, reuse should not be attempted on a large scale at the present time.
- **From the beginning of system development, efforts to plan for reuse should be undertaken by analyzing the system for commonalities that might serve as the basis for reuse.** Such analysis can help in making decisions about when and to what extent software reuse should be applied in WAM.
- **A concept of operations document for software reuse within WAM should be developed by the WAM Program Office.** This document results from the analysis described in the previous recommendation and needs for WAM developed by the WAM Program Office.
- **The state of the art and practice of software reuse should be evaluated periodically to provide input to planning decisions concerning software reuse in WAM.** Particular attention should be paid to techniques for domain analysis and representations, reuse library retrieval systems, and software tools to automate reuse.

- **Other DoD programs involved with software reuse should be investigated and their results made use of in the WAM program where appropriate.** These programs include STARS and SDIO.
- **As early as it appears feasible, a small-scale demonstration project should be initiated in an application in which significant benefits can be expected and for which risk is relatively low.** The demonstration project should be focused not on reuse generally, but on the specific applicability of reuse and the effectiveness of software tools in the technical and organizational environment of WAM.
- **Develop approaches to motivate contractor commitment to reuse before reuse is attempted on a large scale.**
- **Legal and contractual barriers to reuse should be investigated and resolved before contracts are let that require software reuse.** Such barriers include intellectual property rights, economic disincentives for reuse and liability questions.

REFERENCES

American National Standards Institute, Inc. 1983. ANSI/MIL-STD-1815A-1983, *Reference manual for the Ada programming language*. New York: ANSI.

American National Standards Institute (ANSI). 1986. ANSI X3.136-1986, *Database language SQL*. New York: ANSI. Also published as ISO 9075 and FIPS PUB 127.

American National Standards Institute (ANSI). 1989. ANSI X3.135-1989, *Database language SQL with integrity enhancements*. New York: ANSI. Also published as ISO 9075-1989 and FIPS PUB 127-1.

American National Standards Institute (ANSI) Board of Standards Review. BSR X3.194-199X, *Draft SQL2 specification*. New York: ANSI.

Association for Computing Machinery (ACM), Special Interest Group on Ada (SIGAda). 1989. *Ada Letters* 9/3 (Spring).

Baldo, James, Jr., editor. 1990. *Reuse in practice workshop summary*. Alexandria, VA: Institute for Defense Analyses. IDA Document D-754 (draft).

Biggerstaff, T. J. and C. Richter. 1987. Reusability framework, assessment, and directions. *IEEE Software* 4 (March): 41-50.

Biggerstaff, T. J. and A.J. Perlis. 1989. *Software reusability. Volume 1: Concepts and models*. New York: ACM Press.

Booch, Grady. 1987. *Software components with Ada: Structures, tools, and subsystems*. Menlo Park, CA: Benjamin/Cummings.

Defense Communications Agency (DCA). November 1989. WWMCCS ADP Modernization (WAM) decision coordinating paper (DCP). Washington, D.C.: DCA.

Defense Science Board (DSB). 1987. *Report of the Defense Science Board Task Force on military software*. Washington, D.C.: DoD.

DeMillo, Richard A., R. J. Martin, Reginald N. Meeson. 1988. *Strategy for achieving Ada-based high assurance systems*. Alexandria, VA: Institute for Defense Analyses. IDA Paper P-2143.

Department of Defense. 1978. Department of Defense requirements for high order programming languages: STEELMAN. Washington, D.C.: DoD.

Department of Defense. 23 May 1975. DoDD 5000.28, *Design to cost*. Washington, D.C.: DoD.

Department of Defense. 1986. DoDD 5000.3, *Test and evaluation*. Washington, D.C.: DoD.

Department of Defense. 1985. DOD 5200.28-STD, *Trusted computer system evaluation criteria (TCSEC)*. Washington, D.C.: DoD.

Department of Defense, Joint Chiefs of Staff. 1989. Mission need statement (MNS) for the Joint Operation Planning and Execution System (JOPES). Washington, D.C.: DoD.

Digital Equipment Corporation (DEC). 1990. *POSIX tracking report: Volume 2* Issue 3 (July): 1-4.

Edwards, Stephen H. and James Baldo Jr. 1990. *An approach for constructing reusable software components in Ada*. Alexandria, VA: Institute for Defense Analyses. IDA Paper P-2378 (draft).

Emerging Technologies Group, Inc. 1990. *Open system solutions: An analysis of application environments. Comparison of operating system interfaces. An industry report.*

Evangelist, Michael. 1984. Program complexity and programming style. In *Proceedings of the Computer Data Engineering Conference, Los Angeles, April 1984*, 534-541.

Evangelist, Michael. 1983. Software complexity metric sensitivity to program restructuring rules. *Journal of Systems and Software* 3 (September): 231-243.

Federal Information Processing Standards Publication (FIPS PUB). 1990. FIPS PUB 127-1, *Database language SQL. U.S. Department of Commerce, National Institute of Standards and Technology*. Also published as ISO 9075 and ANSI ANSI X3.135-1989.

Griest, T. and M. Bender. 1989. Limitations on the portability of real time Ada programs. *Proceedings of the ACM SIGAda Tri-Ada'89 Conference, Pittsburgh, PA, October 23-26, 1989*, 474-489. New York: ACM.

Grindley, Peter. 1989. Analysis of the information technology standardization process. In *Proceedings of the international symposium on information technology*

standardization process, Braunschweig, West Germany, 4-7 July 1989, edited by J. Berg and H. Schumny, 99-110. Amsterdam, Netherlands: North-Holland.

Halstead, Maurice H. 1977. *Elements of software science*. New York: North-Holland.

IEEE, Inc. 1983. *IEEE standard glossary of software engineering terminology*. New York: IEEE, Inc.

IEEE, Inc. Technical Committee on Operating Systems of the IEEE Computer Society. 1988. IEEE 1003.1, *IEEE standard portable operating system interface for computer environments*. New York: IEEE, Inc.

IEEE, Inc. Technical Committee on Operating Systems of the IEEE Computer Society. December 6, 1989. IEEE 1003.0, *IEEE guide to POSIX open systems environments, draft 6*. New York: IEEE, Inc.

IEEE, Inc. Technical Committee on Operating Systems of the IEEE Computer Society. January 24, 1990. IEEE 1003.3, *Draft IEEE standard for test methods for measuring conformance to POSIX*. New York: IEE, Inc.

International Standards Organization (ISO). 1988a. ISO DIS 9594-1, *The directory, part 1: Overview of concepts, models, and service*. Also published as CCITT X.500). Vienna, VA: Omnicom, Inc.

International Standards Organization (ISO). 1988b. ISO 8571-1, *File transfer, access, and management. Part 1: General introduction*. Also published as IEEE P1238.2. Vienna, VA: Omnicom, Inc.

Isaak, Jim. 1990. Applications environment profiles: A significant tool for simplifying and coordinating standards efforts. *IEEE Computer* (February): 69-70.

Latour, L. and E. Johnson. 1988. Seer: A graphical retrieval system for reusable Ada software modules. In *Proceedings of the third international IEEE conference on Ada applications and environments, Manchester, NH, 23-25 May 1988*, 105-13. Washington, D.C.: IEEE Computer Society Press.

McCabe, Thomas J. 1976. A complexity measure. *IEEE Transactions on Software Engineering* SE-2/4 (December): 308-320.

McIlroy, M.D. 1969. Mass produced software components. In *Software Engineering*, edited by P. Naur and B. Randell, 138-155. Garmisch, Germany: Nato Science Committee.

McNicholl, D. G. et al. 1988. *Common Ada missile packages-Phase 2 (CAMP-2). Volume I: CAMP parts and parts composition system*. St. Louis, MO: McDonnell Douglas Astronautics Company. AFAL-TR-88-62, Volume I.

Martin Marietta. 1989. *Software engineering guidelines for portability & reuse*. Oak Ridge, TN: Martin Marietta.

Myers, Ware. 1989. Allow plenty of time for large-scale software. *IEEE Software* 6/4 (July): 92-99.

Naecher, Philip A. June 1990. POSIX and portability. *DEC Professional* 9 (June): 46-51.

National Institute of Standards and Technology (NIST). 1990. *Open system standards: A Federal strategy*. Gaithersburg, Maryland: NIST.

Nissen, J. and P. Wallis. 1984. *Portability and style in Ada*. Cambridge, UK: Cambridge University Press.

Operating Systems Standards Working Group (OSSWG). 1 June 1990. *Recommendation report for the next-generation computer resources (NGCR) operating systems interface standard baseline*. Compiled by D.P. Juttelstad, Naval Underwater Systems Center (NUSC). NUSC Technical Document 6902.

Pappas, F. 1987. Ada portability guidelines. Waltham, MA: SofTech.

Software Productivity Consortium (SPC). 1989. *Ada quality and style—Guidelines for professional programmers*. New York: Von Nostrand Reinhold.

SofTech, Inc. 1984. *Ada portability guidelines*. Waltham, MA: SofTech, Inc.

SofTech, Inc. 1985. *ISEC portability guidelines*. Waltham, MA: SofTech, Inc.

Taft, William H. 1987a. DoD Directive 3405.1, Computer programming language policy. Washington, D.C.: DoD.

Taft, William H. 1987b. DoD Directive 3405.2, Use of Ada in weapons systems. Washington, D.C.: DoD.

Tracz, W. 1989. Where does reuse start? In *Reuse in practice workshop summary*, edited by James Baldo, 36-46. Alexandria, VA: Institute for Defense Analyses. IDA Document D-754.

UniForm. 1990a. *POSIX explored: System interface*. Santa Clara, CA: UniForm.

United States Air Force (USAF). October 1987. *USAF R&M 2000.* Washington, D.C.: USAF, Office of the Special Assistant for R&M.

WWMCCS ADP Technical Users Group (TUG). 1989. *Proceedings of the WWMCCS ADP technical users group, 1989 fall conference, Fairfax, VA, 5-8 December 1989.*

ACRONYMS

ADP	Automated Data Processing
AEP	Application Environment Profile
ANSI	American National Standards Institute
API	Application Program Interface
BSD	Berkeley System Definition
CAMP	Common Ada Missile Packages
CCIS	Command and Control Information System
CCITT	Commite Consultatif International Telephonique et Telegraphique (Consultative Committee for International Telegraph and Telephone)
COTS	Commercial off the Shelf
DBMS	DataBase Management System
DoD	Department of Defense
ECMA	European Computer Manufacturers Association
FIPS	Federal Information Processing Standards
FTAM	File Transfer, Access and Management
HOLWG	High Order Language Working Group
IDA	Institute for Defense Analyses
IEEE	Institute for Electrical and Electronics Engineers
ISO	International Standards Organization
IV&V	Independent Verification & Validation
JIAWG	Joint Interoperable Avionics Working Group
JOPES	Joint Operations Planning and Employment System
POSIX	Portable Operating System Interface for Computer Environments
MNS	Mission of Need Statement
NASA	National Aeronautical and Space Administration
NATO	Northern Atlantic Treaty Organization
NGCR	Next Generation Computer Resources
NIST	National Institute of Standards and Technology

NS/DS	Namespace & Directory Services
OLTP	On-Line Transaction Processing
OSE	Open Systems Environment
OSF	Open Software Forum
PC	Personal Computer
PII	Protocol Independent Interface
RAPID	Reusable Ada Packages for Information Systems Development
RPC	Remote Procedure Call
SDI	Strategic Defense Initiative
SQL	Structed Query Language
STARS	Software Technology for Adaptable, Reliable Systems
SVID	System V Interface Definition
TCSEC	Trusted Computer Security Evaluation Criteria
TIMS	Traditional Interactive Multiuser System
URG	Uniformity Rapporteur Group
V&V	Verification & Validation
WAM	World Wide Military Command and Control System (WWMCCS) Automated Data Processing (ADP) Modernization
WWMCCS	World Wide Military Command and Control System
XTP	X/Open Transaction Processing

APPENDIX A

POSIX POINTS OF CONTACT

Table A-1 POSIX Points of Contact

POSIX Working Group	Chair/Co-Chair	Organization	E-Mail	Telephone #	Fax #
GUIDANCE					
IEEE/CS P1003	Jim Isaak	DBC	isaac@decvax.dec.com	603-884-3634	603-884-3682
P1003.0 A Guide to POSIX-Based Open Systems	Al Hankinson	NIST	ahank@nwe.ncsl.nist.gov	301-975-3290	301-590-0932
	Kevin Lewis	DEC	klewis@gucci.dec.com	202-383-5017	202-383-5024
SYSTEM SERVICES					
P1003.1 Systems Services and C Language Binding	Donn Terry	Hewlett Packard	donn@hpfcla.fc.hp.com	303-229-2367	303-229-2838
P1003.4 Real-Time Extensions	Bill Corwin	Intel Corporation	wmc@indel.hf.intel.com	503-696-2248	503-696-5399
	Mike Cossey	Martin Marietta		615-574-9217	615-574-9955
P1003.6 Security	Dennis Steinauer	NIST	steinauer@ecf.ncsl.nist.gov	301-975-3357	301-590-0932
	Ron Elliot	IBM	elliot@aisvm.vnet.ibm.net	[1]	[1]
UTILITIES					
P1003.2 Shell and Utilities	Hal Jespersen	POSIX Software Group	hlj@posix.com	415-364-3410	415-364-4498
	Don Cragun	Sun	dwc@sun.com	415-336-7487	415-969-9131
P1003.7 System Administration	Steven L. Carter	Bellcore	slc2@sunni.bellcore.com	201-699-6732	201-981-1456
	David Hinnant	BNR, Inc.	uunet!nrl.nt.org!bnrunix!dfh	919-991-9299	
	Martin Kirk	British Telecom Research Lab	ukclarion!mtk!rk	[2]	[2]
P1003.15 Supercomputing Batch Environment Extension					
LANGUAGE BINDINGS					
P1003.x C Bindings					
P1003.5 Ada Bindings	Steve Deller	Verdix Corporation	deller@verdix.com	703-378-7600	703-378-7532
	Jim Lonjers	Unisys	lonjers@pnc.unisys.com	215-648-7525	215-648-7412
	Maj. T. Fong	U.S. Army	tfong@huachuca-emh2.army.com	602-533-2873	602-533-5676
P1003.9 FORTRAN Bindings	John McGrory	Hewlett Packard	mcgrory@iag.hp.com	408-447-0265	
	Michael Hannah	Sandia National Labs	mjhanna@sandia.gov	505-845-8923	
DISTRIBUTED SYSTEMS SERVICES					
P1003.8 Transparent File Access API	Jason Zions	Hewlett Packard	jason@hpcndm.hp.com	604-447-3600	
	David Dodge	Oracle Complex Systems			
P1003.12 Protocol Independent Interfaces (PII) API	Les Wibberley	Chemical Abstracts	lhw25@cas.hinet	614-447-3600	614-447-3715
P1003.XX Namespace & Directory Services (NS/DS)	Lakshmi Arunachalam	Sun			

Table A-1 POSIX Points of Contact					
POSIX Working Group	**Chair/Co-Chair**	**Organization**	**E-Mail**	**Telephone #**	**Fax #**
P1224 X.400 Mail Services API	John Boebinge	DEC		603-881-2956	
P1237 Remote Procedure Call (RPC) API	Ken Hobday	DEC	hobday@rtl.dec	603-881-2214	603-881-1700
P1238 File Transfer, Access and Management (FTAM)	Kester Fong	General Motors		313-947-0561	
WINDOWING					
P1201.1 Application Toolkit API for the X Window System	Sunhil Mehta	Convergent Technologies	sunil@convergent.com	408-435-3487	408-435-5365
P1201.2 Recommended Practice on Driveability	Lin Brown	Sun	lin@sun.com	415-336-5319	415-336-5708
P1201.3 User Interface Management System (UIMS)	Robert Seacord	SEI	rcs@sei.cmu.edu	412-268-6384	
P1201.4 Library API for the X Window System	Alan Weaver	IBM	linear@weaver.austin.ibm.com	512-823-2611	512-823-2728
CONFORMANCE					
P1003.3 Test Methods for Measuring Conformance to POSIX	Roger Martin	NIST	rmartin@swe.ncsl.nist.gov	301-975-3295	301-590-0932
	N. Ray Wilkes	Unisys	nrw@sp7040	801-594-4643	801-594-3827
PROFILES					
P1003.10 Supercomputing AEP	Karen Sheaffer	Sandia National Labs	karen@anl-srpsgw.llnl.gov	415-294-3431	
	Jonathan Brown	Lawrence Livermore	jbrown@nmfecc.llnl.gov	415-423-4157	
P1003.11 Transaction Processing AEP	Elliot Brebner	Unisys	sunetls5000!brebner	612-635-2500	612-635-2003
	Bob Snead	Interactive Systems	bobs@ico.isc.com	303-449-2870	303-449-2876
P1003.13 Real Time Processing AEP	Bill Corwin	Intel Corporation	wmc@littlei.hf.intel.com	503-696-2248	503-696-5399
	Mike Cossey	Martin Marietta		615-574-9217	615-574-9955
P1003.14 Multiprocessing Application Support AEP	Robert Knighten	Encore Computer	knighten@encore.com	508-460-0500	508-485-0709
P1003.XX Traditional Interactive Multiuser System (TIMS) AEP	Donn Terry	Hewlett Packard	donn@hpfcla.fc.hp.com	303-229-2367	303-229-2838

The information in this table was obtained primarily from presentation slides provided by Roger J. Martin of NIST, dated May 10, 1990.

[1] Phone #'s for Ron Elliot (Germany). Phone: 49-(0)(7031)-18-5097 Fax: 49-(0)(7031)-18-5014

[2] Phone #'s for Martin Kirk (U.K.). Phone: +44-473-642518 Fax: +44-473-643019

Bellcore - Bell Communications Research
SEI - Software Engineering Institute

APPENDIX B

POSIX 1003.1 FEATURES

There are several places where the POSIX 1003.1 standard allows implementation-dependent behavior, but fails to place restrictions on the allowed range of behaviors. An example of this is the *stat()*, *fstat()*, and *kill()* functions.

Stat(), *fstat()* and *kill()* are allowed to fail if an implementation provides additional or alternate file access and security controls. The standard does not specify a minimal situation under which these functions are to succeed. However, the standard does specify an intent by stating that the system may deny the existence of the file specified by *path* or a process specified by *pid*. Thus, the assumed intent is for systems that provide additional security. Applications should be aware that these functions may fail if proper access rights do not exist, although, the wording of the standard allows any restrictions to the behavior.

The following is a list of functions from POSIX 1003.1. Each function is listed with a key to indicated how well defined it is in the standard. A brief comment follows any function that warrants extra concern because of possible extensions or multi-semantic behaviors. The list of keys is given below:

OK — Indicates that it is well defined by the standard. If "OK" is followed by a comment, this indicates a caution to the use of the function. For example, many of the user-id functions can return a static data structure; therefore, the information should be copied to a local structure before another call is made to on of these functions.

EX — Indicates that it is well defined by the standard; however, extensions are likely to be provided by a vendor.

MULTI — Indicates that some part of the behavior may or will be defined by the implementation.

Specific extensions are not listed in the this table. It is assumed that any features provided by a vendor that are not defined in the POSIX 1003.1 standard are extensions by definition. Vendor-specific functions and constants are the two most common forms of

extentions. For example, *fcntl()*, *pathconf()*, *sigaction()*, and *sysconf()* are functions where implementation defined constants maybe included.

Table A-1. P1003.1 System Functions

System function	Key	Comments
access()	MULTI	Success for X_OK (execute permission) may be returned even if none of the execute file permission bits are set. (Use *stat()* or *fstat()* functions to check for execute access.)
alarm()	OK	
cfgetispeed()	OK	
cfgetospeed()	OK	
cfsetispeed()	OK	
cfsetospeed()	OK	
chdir()	OK	
chmod()	MULTI	(1) Implementation-defined restrictions may cause the S_ISUID and S_ISGID bits in *mode* to be ignored. (Use *setuid()* and *setgid()* to alter the uid and gid of a process. Do not rely on the semantics of S_ISUID and S_ISGID.) (2) The effect on file descriptors for files open at the time of the *chmod()* function is implementation-defined. (Do not use *chmod* on open files.)
chown()	MULTI	If *path* argument refers to a regular file, the S_ISUID and S_ISGID bits of the file mode shall be cleared upon successful return from *chown*, unless the call is made by a process with appropriate privileges, in which case it is implementation-defined whether those bits are altered. If *chown()* is successfully involked on a file that is not a regular file, these bits may be cleared. (Do not rely on the semantics of S_ISUID and S_ISGID.)

Table A-1. P1003.1 System Functions

System function	Key	Comments
close()	OK	
closedir()	OK	
creat()	OK	
ctermid()	OK	Data returned may be static. (Copy the data to a local structure if needed.)
cuserid()	OK	Data returned may be static. (Copy the data to a local structure if needed.)
dup()	OK	
dup2()	OK	
execl()	EX	Inheritance of process attribute not defined in the standard is implementation-defined.
execle()	EX	Inheritance of process attribute not defined in the standard is implementation-defined.
execlp()	MULTI	File should either contain a "/" or the PATH environment variable should be defined.
	EX	Inheritance of process attribute not defined in the standard is implementation-defined.
execv()	EX	Inheritance of process attribute not defined in the standard is implementation-defined.
execve()	EX	Inheritance of process attribute not defined in the standard is implementation-defined.

Table A-1. P1003.1 System Functions

System function	Key	Comments
execvp()	MULTI	File should either contain a "/" or the PATH environment variable should be defined.
	EX	Inheritance of process attribute not defined in the standard is implementation-defined.
exit()	OK	
fcntl()	EX	Only values listed in the standard should be used for *cmd*.
fork()	OK	
fpahtconf()	EX	Only use the variables defind in Table 5-2 and the restrictions given in the standard.
fstat()	MULTI	Implementations that provided additional or alternate file access control mechanisms, may under implementation-defined conditions, cause this function to fail. (Applications should handle all error conditions.)
getcwd()	OK	
getegid()	OK	
getenv()	OK	
geteuid()	OK	
getgid()	OK	
getgrgid()	OK	Data returned may be static. (Copy the data to a local structure if needed.)

Table A-1. P1003.1 System Functions

System function	Key	Comments
getgrnam()	OK	Data returned may be static. (Copy the data to a local structure if needed.)
getgroups()	OK	
getlogin()	OK	Data returned may be static. (Copy the data to a local structure if needed.)
getpgrp()	OK	
getpid()	OK	
getppid()	OK	
getpwnam()	OK	Data returned may be static. (Copy the data to a local structure if needed.)
getpwuid()	OK	Data returned may be static. (Copy the data to a local structure if needed.)
getuid()	OK	
isatty()	OK	
kill()	MULTI	The behavior is undefined if *pid* = -1. Implementation that provide extended security controls may impose implementation-defined restrictions on the sending of signals. (Applications should handle all error conditions.)
link()	OK	
lseek()	OK	
mkdir()	OK	
mkfifo()	OK	

Table A-1. P1003.1 System Functions

System function	Key	Comments
open()	OK	
opendir()	OK	
pathconf()	EX	Only use the variables defind in Table 5-2 and the restrictions given in the standard.
pause()	OK	
pipe()	OK	
read()	OK	
readdir()	OK	
rename()	MULTI	*rename()* on directories should have write permission. (Do not rely on the ability to rename directories without having write permission.)
rewinddir()	OK	
rmdir()	OK	
getgid()	OK	
setpgid()	OK	
setsid()	OK	
setuid()	OK	
sigaction()	EX	Only use actions defined in the standard. Only functions defined as "safe" by the standard should be called from signal handlers.
sigaddset()	OK	
sigdelset()	OK	
sigemptyset()	EX	Only signals defined in the standard are required to be empty.

Table A-1. P1003.1 System Functions

System function	Key	Comments
sigfillset()	EX	Only signals defined in the standard are required to be filled.
sigismember()	OK	
sigpending()	OK	
sigprocmask()	MULTI	If SIGFPE, SIGILL, and SIGSEGV are generated while blocked the result is not defined by the standard, unless they were generated by *kill()* or *raise()* of C.
sigsuspend()	OK	
sleep()	MULTI	Signal catching should not alter the schedule for SIGALRM or BLOCK SIGALRM; calling *siglongjmp()* or *longjmp()* to restore the environment to a state prior to *sleep* results in undefined behavior. (Do not block SIGALRM or alter the schedule of SIGALRM.)
stat()	MULTI	Implementations that provided additional or alternate file access control mechanisms, may under implementation-defined conditions, cause this function to fail. (Applications should handle all error conditions.)
sysconf()	EX	Only use names defined in the standard.
tcdrain()	OK	
tcflow()	OK	

Table A-1. P1003.1 System Functions

System function	Key	Comments
tcflush()	OK	
tcgetattr()	OK	
tcgetpgrp()	OK	
tcsendbreak()	OK	
tcsetattr()	OK	
tcsetpgrp()	OK	
time()	OK	
times()	EX	Only use the structure part defined in the standard.
ttyname()	OK	Data returned may be static. (Copy the data to a local structure if needed.)
umask()	OK	
uname()	OK	Format is implementation defined.
unlink()	MULTI	Should used *rmdir()* to remove a directory.
utime()	OK	
wait()	EX	Child processes should not use extensions.
waitpid()	EX	Child processes should not use extensions.
write()	OK	

APPENDIX C

ANSI SQL IMPLEMENTATION DEPENDENCIES

This appendix examines implementor-defined aspects of Sections 1 through 8 of the ANSI SQL standard. Each applicable instance of the term "implementor-defined" is denoted by its section number, the name of the section, and the page number that section is found within the ANSI SQL standard. The sentence containing the word "implementor-defined" is set in italic font. If the sentence is found within a context paragraph, that paragraph is set in Roman font. A recommendation is made following each "implementor-defined" aspect as to its effect on portability.

1. **Section 3.3 Conventions, page 6**

   In the Syntax Rules, the term "shall" defines conditions that are required to be true of syntactically conforming SQL language. *The treatment of SQL language that does not conform to the Formats or the Syntax Rules is implementor-defined.*

   Recommendation: This paragraph states that supersets to the ANSI SQL standard are allowed, but does not directly affect portability of existing SQL constructs.

   In the General Rules, the term "shall" defines conditions that are tested at run-time during the execution of SQL statements. If all such conditions are true, then the statement executes successfully and the SQLCODE parameter is set to a defined nonnegative number. *If any such condition is false, then the statement does not execute successfully, the statement execution has no effect on the database, and the SQLCODE parameter is set to an implementor-defined number.*

   Recommendation: SQL applications should be written such that the specific implementation-dependent code affected by the SQLCODE parameter is appropriately localized or abstracted such that future changes in the result of the SQLCODE do not require substantial changes to the application.

2. **Section 3.3 Conventions, page 6**

   The term "persistent object" is used to characterize objects such as <module>s and <schema>s that are created and destroyed by implementor-defined mechanisms.

Recommendation: This implementor-defined aspect of ANSI SQL does not significantly affect application portability.

3. **Section 4.2 Data types, page 9**

   A data type is a set of representable values. The logical representation of a value is a <literal>. *The physical representation of a value is implementor-defined.*

   A null value is an implementor-defined type-dependent special value that is distinct from all nonnull values of that type.

4. **Section 4.2.1 Character strings, page 9**

   A character string consists of a sequence of characters of the implementor-defined character set.

   Recommendation: Unless a requirement exists to the contrary, SQL implementations should provide support for the 95-character graphic subset of ASCII (FIPS PUB 1-2).

5. **Sectior 4.5 Integrity constraints, page 11**

   Integrity constraints are effectively checked after execution of each <SQL statement>. *If the base table associated with an integrity constraint does not satisfy that integrity constraint, then the <SQL statement> has no effect and the SQLCODE parameter is set to an implementor-defined negative number.*

   Recommendation: SQL applications should be written such that the specific implementation-dependent code affected by the SQLCODE parameter is appropriately localized or abstracted such that future changes in the result of the SQLCODE do not require substantial changes to the application.

6. **Section 4.6 Schemas, page 11**

   A <schema> is a persistent object specified by the schema definition language. It consists of a <schema authorization clause> and all <table definitions>s, <view definition>s, and <privilege definition>s known to the system for a specified <authorization identifier> in an environment. *The concept of environment is*

implementor-defined.

Recommendation: This implementor-defined aspect of ANSI SQL does not significantly affect application portability.

7. **Section 4.7 The database, page 11**

The database is the collection of all data defined by the <schema>s in an environment. *The concept of environment is implementor-defined.*

Recommendation: This implementor-defined aspect of ANSI SQL does not significantly affect application portability.

8. **Section 4.8 Modules, page 11**

An application program is a segment of executable code, possibly consisting of multiple subprograms. A single <module> is associated with an application program during its execution. An application program shall be associated with at most one <module>. *The manner in which this association is specified, including the possible requirements for execution of some implementor-defined statement, is implementor-defined.*

9. **Section 4.12 Cursors, pages 12-13**

A cursor in the open state designates a table, an ordering of the rows of that table, and a position relative to that ordering. *If the <declare cursor> does not specify an <order by clause>, then the rows of the table have an implementor-defined order.* This order is subject to the reproducibility requirement within a transaction (see 4.16 "transactions" on page 14), but it may change between transactions.

If a cursor is before a row and a new row is inserted at that position, then the effect, if any, on the position of the cursor is implementor-defined.

If an error occurs during the execution of an <SQL statement> that identifies an open cursor, then the effect, if any, on the position or state of that cursor is implementor-defined.

A working table is a table resulting from the opening of a cursor. *Whether opening a cursor results in creation of a working base table or a working viewed table is*

implementor-defined.

10. **Section 5.1 <character>, Syntax Rules, page 15**

1) A <special character> is any character in the implementor-defined character set other than a <digit> or a <letter>. If the implementor-defined end-of-line indicator is a character, then it is also excluded from <special character>.

11. **Section 5.3 <token>, Format, page 19**

<newline> ::= implementor-defined end-of-line indicator

12. **Section 5.5 <data type>, Syntax Rules, pages 22-23**

3) If a <length> is omitted, then it is assumed to be 1. If a <scale> is omitted, then it is assumed to be 0. *If a <precision> is omitted, then it is implementor-defined.*

Recommendation: SQL applications should always specify a required <datatype> <precision>.

7) DECIMAL specifies the data type exact numeric, with the scale specified by the <scale> and with implementor-defined precision equal to or greater than the value of the specified <precision>.

Recommendation: This implementor-defined aspect of ANSI SQL does not significantly affect application portability.

8) INTEGER specifies the data type exact numeric, with implementor-defined precision and scale 0.

9) SMALLINT specifies the data type exact numeric, with scale 0 and implementor-defined precision that is not larger than the implementor-defined precision of INTEGER.

Recommendation: This implementor-defined aspect of ANSI SQL does not significantly affect application portability.

11) REAL specifies the data type approximate numeric, with implementor-defined precision.

12) DOUBLE PRECISION specifies the data type approximate numeric, with implementor-defined precision that is greater than the implementor-defined precision of REAL.

Recommendation: This implementor-defined aspect of ANSI SQL does not significantly affect application portability.

13. **Section 5.6 <value specification> and <target specification>, Syntax Rules, page 24**

2) A <parameter specification> identifies a parameter or a parameter and an indicator parameter. The data type of an indicator parameter shall be exact numeric with a scale of 0. *The specific <exact numeric type> of indicator parameters is implementor-defined.*

3) A <variable specification> identifies a host variable or a host variable and an indicator variable. *The data type of an indicator variable shall be the implementor-defined data type specified for indicator parameters.*

6) The data type USER is character string of implementor-defined length.

14. **Section 5.8 <set function specification>, Syntax Rules, pages 27-28**

7) If COUNT is specified, then the data type of the result of a <set function specification> is exact numeric with implementor-defined precision and scale 0.

9) If SUM or AVG is specified, then:
a) T shall not be character string.

b) If SUM is specified and T is exact numeric with scale S, then the data type of the result is exact numeric with implementor-defined precision and scale S.

c) If AVG is specified and T is exact numeric, then the data type of the result is exact numeric with implementor-defined precision and scale.
d) If T is approximate numeric, then the data type of the result is

approximate numeric with implementor-defined precision.

15. **Section 5.9 <value expression>, Syntax Rules, page 29**

4) If the data type of both operands of an operator is exact numeric, then the data type of the result is exact numeric, with precision and scale determined as follows:

a) Let s1 and s2 be the scale of the first and second operands respectively.

b) The precision of the result of addition and subtraction is implementor-defined, and the scale is max(s1, s2).

c) The precision of the result of multiplication is implementor-defined, and the scale is s1+s2.

d) The precision and scale of the result of division is implementor-defined.

5) If the data type of either operand of an operator is approximate numeric, then the data type of the result is approximate numeric. *The precision of the result is implementor-defined.*

16. **Section 5.11 <comparison predicate>, General Rules, page 32**

6) Two strings are equal if all <character>s with the same ordinal position are equal. If two strings are not equal, then their relation is determined by the comparison of the first pair of unequal <characters> from the left end of the strings. *This comparison is made with respect to the implementor-defined collating sequence.*

17. **Section 6.1 <schema>, Syntax Rules, page 53**

1) The <schema authorization identified> shall be different from the <schema authorization identifier> of any other <schema> in the same environment. *The concept of environment is implementor-defined.*

Recommendation: This implementor-defined aspect of ANSI SQL does not significantly affect application portability.

18. **Section 6.5 <view definition>, General Rules, page 57**

1) A <view definition> defines a viewed table. The viewed table, V, is the table that would result if the <query specification> were executed. *Whether a viewed table is materialized is implementor-defined.*

19. **Section 7.2 <module name clause>, Syntax Rules, page 62**

1) The <module name> shall be different from the < module name> of any other <module> in the same environment. *The concept of environment is implementor-defined.*

Recommendation: This implementor-defined aspect of ANSI SQL does not significantly affect application portability.

20. **Section 7.3 <procedure>, Syntax Rules, pages 64-5**

8) Case:

a) If the subject <language clause> specifies COBOL, then:

i) The type of the SQLCODE parameter shall be COBOL usage COMPUTATIONAL picture S9(PC), where PC is an implementor-defined precision that is grater than or equal to 4.

d) If the subject <language clause> specifies PL1, then:

i) The type of the SQLCODE parameter shall be PL/I FIXED BINARY (PP), where PP is an implementor-defined precision that is greater than or equal to 15.

21. **Section 7.3 <procedure>, General Rules, page 65**

1) A <procedure> defines a procedure that may be called by an implementor-defined agent.

3) Case:

b) If S did not execute successfully, then:

i) The SQLCODE parameter is set to a negative number whose value is implementor-defined.

Recommendation: SQL applications should be written such that the specific implementation-dependent code affected by the SQLCODE parameter is appropriately localized or abstracted such that future changes in the result of the SQLCODE do not require substantial changes to the application.

22. **Section 8.3 <declare cursor>, General Rules, page 71**

3) Case:

a) If ORDER_BY is not specified, then the ordering of rows in T is implementor-defined. This order is subject to the reproducibility requirements within a transaction, but it may change between transactions.

b) If ORDER BY is specified, then T has a sort order:

vii) Ordering is determined by the comparison rules specified in 5.11, "<comparison predicate>" on page 32. *The order of the null value relative to nonnull values is implementor-defined, but shall be either greater than or less then all nonnull values.*

23. **Section 8.6 <fetch statement>, General Rules, page 74-75**

5) *The assignment of values to targets in the <fetch target list> other than the SQLCODE parameter is in an implementor-defined order.* The SQLCODE parameter is assigned a value last.

6) If an error occurs during the assignment of a value to a target, then the SQLCODE parameter is set to an implementor-defined negative number, and the values of targets other than the SQLCODE parameters are implementor-defined.

24. **Section 8.10 <select statement>, General Rules, page 82**

4) *The assignment of values to targets in the <select target list> other than the SQLCODE parameter is in an implementor-defined order.* The SQLCODE parameter is assigned a value last.

5) If an error occurs during the assignment of a value to a target, then the SQLCODE parameter is set to a negative number whose value is implementor-defined, and the values of targets other than the SQLCODE parameter are implementor-defined.

APPENDIX D

ADA PORTABILITY GUIDELINES

1. INTRODUCTION

This appendix presents an assessment of each of the seven Ada portability guidelines reviewed during this fast reaction task. The approach taken in developing the guidelines as well as the structure of the guidelines can be used by the software developer in determining which guidelines may be most suitable for a particular project. The strengths and weaknesses of each guideline may be used to either select or modify the use of a specific guideline.

2. ADA PORTABILITY GUIDES

2.1 NISSEN - PORTABILITY AND STYLE IN ADA

2.1.1 Approach

Portability and Style in Ada was the first guideline that was published on the subject of Ada portability. It was produced in 1980 by the Ada-Europe Portability Working Group under the chairmanship of John Nissen and edited by Peter Wallis. The guide represents an effort to establish and present guidelines in concert with the Ada language standardization process. This is the aspect that sets this guide apart from the ones that followed it.

The guide proposes a set of rules to be used as advice for programmers or compiler implementors. The emphasis is on the pragmatic use of these rules as opposed to a rigorous, restricting application of requirements. Therefore, a certain amount of flexibility allows the user to weigh the implications in using certain optional or language-dependent features against the cost of minimizing or eliminating their use by programming for portability.

2.1.2 Overview

The guide starts off with a discussion of the need for standards. The availability and use of standards is critical in its assistance to portability. Without standards, the risk of significantly differing implementations of the same language is great. Portability is less likely to be achieved when dealing with implementations that have significant differences and dependencies. In an attempt to give portability as much language support from the start, Ada was standardized before any compilers were written. To ensure complete conformity to the Ada standard, compilers are rigorously tested and certified. Even so, there are allowances for certain options and dependencies in Ada. Those items require careful consideration and treatment with regard to their effect on portability.

Since the guide was developed in concert with the standardization of Ada, it is organized around the Ada standard [ANSI 1983]. The section numbering parallels the standard's chapter/section numbering to the fullest extent possible. Items of issue are discussed at their first reference even if their complete treatment in the standard occurs much later. This was done to evenly distribute the guidance over the whole of the

standard. Not all of the standard's sections are discussed; those with no issues are skipped. For the sections that are discussed, the following format is used:

- Classification. The authors' judgement of the importance of adherence to a rule. There are three classifications:
  - — Mandatory. The given rule when using this language feature must be followed explicitly.
  - — Recommended. The given rule when using this language feature may be partially followed or broken, but not without documenting the justification in the code.
  - — Suggested. The given rule is provided as an aid to portability. Its lack of application will not negatively affect portability.
- Aid. Test procedures, software engineering tools, or other items that, if available, would be of assistance.
- Information from implementors. Additional information about optional features or implementation dependencies.
- Target Requirements. Specific characteristics of the target that should be documented.
- Note(s). Any further relevant information.

In addition, the guide enumerates certain minimum values which are assumed that any target implementation has.

2.1.3 Strengths

Overall, this is an excellent guide to developing or modifying software to be portable. The strengths of this guide fall into four areas:

- Produced in concert with the Ada standard. By developing the guide as the Ada language was being standardized, the authors' were able to gain maturity for the guide. This maturity is reflected in the concise presentation and the comprehensiveness in the coverage of the issues of portability raised by the language. Also, as the first guide, the impact upon the first compilers and subsequent Ada-based systems was tremendous.
- Longevity. Being the first guide loses its importance if the guide is ignored or superceded. This guide is not ignored and although other guides exist, in our opinion this guide has not been superceded. In fact, most of the other guides examined use this one as a basis.
- Guide format. The format chosen for the guide makes it easy to use. The

classifications immediately let the programmer know if this language feature needs to receive special consideration and the other information blocks provide additional guidance.

- Examples. The guide uses Ada code to illustrate certain points. This enhances the understandability and usefulness of the guide.

2.1.4 Weaknesses

The only weakness of this guide is that its longevity also means that it carries the flaws from the past. Although there is only one Ada standard, official interpretations of the standard to resolve conflicts, provide clarity, or correct errors, are not taken into account by this guide since it is not a dynamic document. A good example of an official interpretation is the virtual elimination of the pre-defined exception NUMERIC_ERROR, replaced by an expanded use of CONSTRAINT_ERROR. The guide, understandably, does not reflect this change. A recommendation for using this guide would include scanning the approved interpretations, released as Ada Issues by the Ada Joint Program Office (AJPO) [ACM SIGAda 1989], for those that may affect the validity of the guide.

2.2 SOFTECH - ADA PORTABILITY GUIDELINES

2.2.1 Approach

The *Ada Portability Guidelines* were produced by SofTech, Waltham, MA and submitted to the U.S. Air Force Systems Command, Electronic Systems Division, Hanscom AFB, MA in September 1984 [SofTech 1984].

Two other versions of these guidelines exist. The *Ada Portability Guidelines* produced by Frank Pappas at SofTech in March 1985 were submitted to the U.S. Air Force Electronic Systems Division, Hanscom AFB, MA. The contents of this guideline are identical in content but differ in presentation to the SofTech *Ada Portability Guidelines* [SofTech 1984].

The *ISEC Portability Guidelines* were produced by SofTech in December 1985 [SofTech 1985] and submitted to the U.S. Army Information Systems Engineering Command, Hanscom AFB, MA. As with the Pappas guidelines, the contents are virtually identical to [SofTech 1984].

The approach taken by SofTech [SofTech 1984] was to review the work by Nissen and Wallis [Nissen 1984] and generate a self-contained guide to writing portable Ada programs.

The thesis proposed by the authors is that software portability is characterized by three goals: program behavior, source modification, and safety. The SofTech guidelines are written to specifically address these goals and how the constructs of the Ada programming language impact them. The software developer then performs tradeoff analyses using these goals to determine whether an individual guideline should be applied or disregarded.

The authors of the SofTech guidelines define program behavior as the way the software functions when running in different environments. The primary goal is for the program to exhibit identical behavior in any environment. A second goal in developing portable software is to eliminate the need for source code modification when the software is moved to a different environment. That is, the amount of change required is an indication of the portability. A third goal in developing portable software is safety. Safe software will provide early warnings of any problems when moved to a different environment. That is, warnings of incompatibilities will occur during compilation rather than unexpected failures occurring during execution.

The SofTech guidelines were developed after a thorough review of the Nissen and Wallis guidelines. SofTech provides a brief analysis of this work and a description of its inadequacies. This discussion is the basis for the approach taken in the development of the SofTech Guidelines. The fundamental flaw with the Nissen and Wallis work as described by SofTech is that the guidelines provide no explanation of how using or avoiding a specific feature of the Ada programming language will enhance or degrade portability. The guidance only takes the form of a scale of rules ranging from mandatory to notes with no information on the tradeoffs among behavior, modification, and safety characteristics that can be made in deciding whether to use a specific Ada feature.

2.2.2 Overview

Following an introduction stating the objectives of the project, the *Ada Portability Guidelines* provide a brief explanation of the goals that may be used to achieve portability, namely source modification, program behavior, and safety. This explanation forms the basis for the structure of the guidelines. That is, individual features of the Ada programming language are considered with regard to their impact on these goals.

The next section of the document briefly describes the Ada-Europe Guidelines developed by Nissen and Wallis. The inadequacies of the Nissen and Wallis guidelines are discussed as the motivation for developing an improved set of guidelines.

Section 4 describes what the guidelines are and how they are useful to both developers and managers. SofTech has assembled guidelines that they claim can be used by

developers to build portable Ada programs and by individuals trying to understand the issues underlying portability. The guidelines are written to provide the developer with information about how the use of specific features will impact the program behavior, source modification, and safety during porting. In addition the guidelines offer ideas and provide references addressing the use of inherently less portable features of Ada. A moderate number of examples are provided in this guideline.

The next section addresses the issue of education, style guides, and automated tools to support the development of portable Ada code. SofTech indicates that to write portable Ada code a developer must have a good understanding of Ada. This point cannot be overemphasized. For this reason, the guidelines alone will not guarantee portable code. Sufficient resources and time must be put into Ada education. Further, more than just a knowledge of the Ada programming language is necessary. Some appreciation for the types of optimizations made by the compiler, options open to the run time system, and the underlying machine are also necessary for the developer to understand the opportunities for non-portability to be written into a design. Experience with the language will provide designers with a greater sense of the issues surrounding portability as well as an information store of ideas for dealing with them. A number of references are provided at the end of this section.

The actual guidelines section of the document begins with a short chapter describing what portability is and why it is important. The potential for non-portability is discussed through the use of two examples which give the reader only a vague notion of the ramifications of language features and the underlying target. The three aspects of portability (program behavior, source modification and safety) are discussed briefly. These provide the basis for the tradeoffs made during development of portable software. A substantial effort is made to illustrate the portability tradeoffs that are possible for a given situation. To do this, a single example is used. The example involves the use of integer arithmetic on machines with different integer implementations. Possible approaches to designing portable code are discussed. Each approach is discussed in terms of its impact on the goals of portability. References are provided in this section to potential solutions to some of the portability problems.

SofTech points out that stylistic considerations such as line length can be handled by automated tools and need not be considered portability issues.

The use of other tools, such as portability analyzers, to assist the developer in generating portable code is also discussed. The role of such a tool would be to assist the developer in creating portable code by flagging potential language uses that may lead to portability problems.

The *Ada Portability Guidelines* do not consider incorrect order dependencies and erroneous programs, which received treatment throughout the Nissen and Wallis guidelines as though they were portability issues, to be portability issues. These issues are summarized at the beginning of the guidelines in Chapter 1, and are not treated further.

Stylistic considerations may impact software portability; following a standard convention may facilitate the porting process. Following the guidance provided in a style guide can impact software developer portability substantially. Time can be wasted coming up to speed, or understanding an existing piece of software. By using programming style conventions, this can be minimized. A brief list of style guides is provided.

The structure of the style guide section of the SofTech document follows that of the Ada standard. That is, each chapter of the standard is reviewed for its impact on the three main portability goals: program behavior, modification, and safety. In each section, the tradeoffs that a designer may make are pointed out and a rationale given for the selection of any particular solution. Workarounds to some portability problems are suggested. Each section concludes with a recommendation for the treatment of the Ada language features covered in the parallel chapter of the standard.

Some sections of the Ada standard contain language features that do not impact portability. Discussion of these language features is simply omitted from the *Ada Portability Guidelines*.

2.2.3 Strengths

- An analysis of the meaning of software portability. The Guidelines have gone beyond a careful review of the language features presented in the Ada standard. SofTech has made an attempt to distill the essence of portability in their identification of the three goals of portability: program behavior, source modification, and safety. While these three goals may not capture every aspect of portability, they give software developers a framework in which a tradeoff analysis can occur. This must certainly be an improvement over the random selection of features and implementations that typically occurs during development.
- Parallel structure with the Ada standard. The *Ada Portability Guidelines* follow the structure of the Ada standard. However, the discussion of the Ada programming language features occurs within the analysis of the goals to be achieved when designing portable software. Each language feature is examined for its impact on program behavior, the need for source code modification, and safety. As the features are discussed, the rationale for an Ada language feature is given within the context of a tradeoff among the portability goals. Thus, the guidelines support the software developer in making the important

decisions. The guidelines generally do not provide rigid rules that a developer is instructed to follow if portability is to be achieved. Rather, they provide the information that the developer needs to perform a tradeoff analysis of the options available. These guidelines require that the software developer is an experienced Ada practitioner. However, without an understanding of Ada, even rigid rules for portability would not guarantee success.

2.2.4 Weaknesses

- The claim that the guidelines can educate someone on the issues of portability is not supported in the document. The Guidelines are not a tutorial on portability. Although examples and counter examples are given, insufficient background is given on the issues of how compilers, run time systems, and machine architectures impact the behavior of the program.
- Lack of guidance for the software developer generating real-time applications. Real-time applications require additional guidance. This is not provided or even acknowledged in the *Ada Portability Guidelines*.
- As is the case with *Portability and Style in Ada* by Nissen and Wallis, this document needs to be updated to comply with decisions reached in the AJPO Ada Issues [ACM SIGAda 1989].

2.3 MARTIN MARIETTA - SOFTWARE ENGINEERING GUIDELINES FOR PORTABILITY AND REUSABILITY

2.3.1 Approach

The *Software Engineering Guidelines for Portability and Reusability* was developed by Martin Marietta. This guide differs from the other guides in that it uses an actual case study in its presentation. The Software Development and Maintenance Environment (SDME) was envisioned to function as an Ada Programming Support Environment (APSE) for WIS software. In the future WIS, the underlying system was to be POSIX compliant. However, when the SDME project was initiated, a POSIX-compliant system for WWMCCS hardware was not in place. Therefore, a VAX VMS system became the base system for SDME. The later porting of SDME from VMS to a POSIX-compliant system provided the case study used in this guide.

The position that this guide takes with regard to portability is that portability is a subset of the more general idea of reusability. The rationale is that when discussing portability, the focus is on "reusing" a complete system or subsystem on a platform different from which it currently resides. It follows then that reusability is the porting of generalized routines in current systems to create new systems. While other guides state that enhancing

software portability will in turn enhance reusability, the approach taken by this guide ensures that portability is an integral part of reusability.

2.3.2 Overview

The structure of this guide follows a list of seven general areas with specific ideas applicable to portability. The areas loosely conform to major chapter headings within the Ada standard and represent a consolidation of ideas from the other guides mentioned in previous sections. The seven areas are:

- Fundamentals. This area covers some basic aspects, such as non-standard character sets, documentation, and order dependencies, as they affect portability.
- Numeric Types and Expressions. This area addresses some aspects of numeric types and expressions built from objects of those types, such as which model is used for floating point and the evaluation of subexpressions, as they affect portability.
- Storage Control. This area specifically focuses on dynamic structures management.
- Tasking. This area addresses some aspects of tasking, such as delay statement use and select statement evaluation, that affect portability.
- Generic Units. This area should be dealing with those aspects of portability affected by the use of generic units. However, the ideas stated for discussion deal with exception handling; generic units are not mentioned.
- Representation Clauses and Implementation-Dependent Features. This area addresses some aspects relating to the underlying system, such as interfacing to other languages and implementation-defined pragmas, that affect portability.
- Input/Output (I/O). This area addresses some aspects of I/O, such as implementation specific I/O, that affect portability.

2.3.3 Strengths

The strengths of this guide are:

- The consolidation of ideas rather than the rote following of the Ada standard. By grouping across concepts instead of simply by syntactic construction, this guide provides a pragmatic treatment of portability. For example, in the first area, Fundamentals, the ideas of "custom bodies" and "port verification" do not correspond to a particular Ada standard reference, but are important considerations when porting software.
- The use of an actual case history to support its recommendations. The SDME

porting from a VAX VMS environment to a POSIX-compliant environment provides an excellent source from which to translate "lessons learned" into recommendations for the future. For example, the idea of interfacing Ada with other languages might not be considered risky when presented in the Ada standard. That is, if these instances are isolated and documented then there will be less problems if the code is ported. This however, is not true. The experience of the SDME porting showed that minimizing the amount of foreign code along with isolating and documenting it provides the best results.

2.3.4 Weaknesses

The only weakness is the lack of coded examples to illustrate some of the SDME experience. The descriptions are occasionally vague, for example, in the section Generic Units where the ideas discussed refer to exception handling only; generic units are never mentioned. The exceptional conditions described would certainly affect generic units, but an explicit explanation as to how and an example of code would be helpful to the less experienced Ada programmer.

2.4 SPC - ADA QUALITY AND STYLE - GUIDELINES FOR PROFESSIONAL PROGRAMMERS

2.4.1 Approach

The Software Productivity Consortium's (SPC) *Ada Quality and Style - Guidelines for Professional Programmers* [SPC 1989] was intended to provide guidelines to generally improve the quality of the Ada software being produced. The book does not focus specifically on developing portable or reusable software, or on any one particular software domain such as real-time, embedded software. However, it does address these issues as part of the goal of assisting developers in producing better Ada software.

The book is aimed at three classes of software personnel: programmers new to Ada, programmers experienced with Ada, and managers of software development efforts. A separate chapter is dedicated to the role of the manager in producing higher quality Ada code.

Through discussions of issues and examples of code the book provides specific guidance for using individual features of the Ada programming language. The rationale for the guidance is also provided. In some cases the guidance may be legitimately disregarded. A discussion of these exceptions is intended to give the developer some latitude in making design decisions.

This book was not written as an introductory text on Ada and references to introductory texts are provided. The reader is encouraged to become familiar with the material in these introductory texts as well as with the Ada standard. The guidelines contained in the book are cross-referenced to the Ada standard.

2.4.2 Overview

The organization of *Ada Quality and Style - Guidelines for Professional Programmers* does not rigidly follow the structure of the Ada standard as do many of the guidelines previously developed. Rather, a functional approach to improved software quality has been adopted. That is, specific issues that may impact software quality are discussed in terms of the Ada language features that may apply.

The book begins with general guidance to the three classes of software personnel. This guidance is more an overview of what each class of individuals may contribute toward the goal of increased software quality. References that may be of particular interest to each class are given.

Chapters 2-9 cover the following topics:

- Source code and presentation
- Readability
- Program structure
- Programming practices
- Concurrency
- Portability
- Reusability
- Instantiation

Each chapter begins with a general discussion of the topic as it relates to software quality. Following the introductory remarks, the chapter is broken down into sections that address the issues related to the topic. The material in these sections is presented under the following headings: guideline, example, rationale, exceptions, and notes.

Guidelines take the form of stylistic information or rules for using or avoiding specific features of the Ada language. In some cases, a guideline is associated with additional reference material. The guidelines are not individually numbered but are simply listed under the issue being addressed.

Examples of Ada code are only provided for selected guidelines. Some of the examples intentionally contain errors to illustrate the incorrect use of a language feature. In other cases the examples indicate the correct syntax and use of the Ada feature.

The rationale for the guideline provides justification for the guidance. This section frequently includes references to the Ada standard or other work that supports the conclusions formulated by the authors. References to other guidelines are also provided where appropriate. The information in the rationale section allows a developer to assess the relevance of the guidance to the domain.

Situations in which the developer may elect not to apply specific guidelines are intended to be rare. These exceptional cases are described under the exceptions heading of the section. The risks or benefits that the software developer can expect from disregarding the guideline are also discussed.

Additional miscellaneous information is provided under the heading of notes. This information may elaborate or clarify material presented under another heading.

Chapter 10 provides the software developer with a complete example of code developed using the guidelines. The code is preceded by a short explanation of its function and some notes on its design.

Appendix A of [SPC 1989] is a cross reference between the guidelines and the Ada standard.

2.4.3 Strengths

- The structure of *Ada Quality and Style - Guidelines for Professional Programmers* may be considered both a strength and a weakness. The format of following a guideline by an example, rationale, exceptions and finally notes is useful. Guidance should provide enough information to allow the designer to make a decision as to whether to apply or disregard a guideline. Blindly following rules without understanding the implications would be dangerous. Including a complete example of software developed using the guidelines is valuable. A small piece of code, taken out of context sometimes fails to illustrate the point being made. The reference and bibliography lists provide the software developer with pointers to most if not all that has been written on programming in Ada. The guidelines are organized functionally rather than by language construct. This approach is logical until a developer needs guidance on a specific Ada feature. Then, instead of going directly to the text, the developer must go to Appendix A - the Ada standard cross reference map. Appendix A maps the coverage of Ada language constructs to one or more chapters of the text.
- The guidelines acknowledge the role of management as well as different levels of programmers in producing high quality Ada software. An entire chapter is dedicated to the role of management in the development process.

Management as well as the contracting process are critical to achieving the goals of portability and reusability.

- The guidelines provided in this book build on a large body of previous work. A great deal has been written about the use of the Ada programming language. Several portability and style guides exist. This book contains many of the same recommendations that can be found in these other books and reports. The authors acknowledge, for instance, that the material in the portability chapter was "largely acquired" from these sources. Appropriately, the authors provide pointers to these references.

2.4.4 Weaknesses

- The lack of substantial new guidance based on experience gained since the first guidelines on Ada portability were written.
- Attributes resulting from the use or avoidance of specific features are only sometimes mentioned. The attributes of interest include, but are not limited to, performance, behavior, and source modification.
- The guide would benefit from additional examples.

2.5 GRIEST - LIMITATIONS ON THE PORTABILITY OF REAL-TIME ADA PROGRAMS

2.5.1 Approach

Limitations on the Portability of Real-Time Ada Programs is a set of guidelines for developing portable real-time Ada applications. They are the result of a study of real-time portability issues conducted at the U.S. Army Center for Software Engineering at Ft. Monmouth, New Jersey.

Several Ada portability guidelines were already in existence when the Center for Software Engineering began research on Ada portability issues. However, these existing guidelines all assumed that to achieve portability, implementation-dependent features of Ada would be avoided. The development of real-time applications in Ada requires the use of many of these implementation dependent features, making the use of existing guidelines inappropriate. The goal of the work at the Center for Software Engineering was to provide the developer of real time systems with guidelines for producing portable code.

Implementation dependencies in the Ada programming language were analyzed. The research involved studies on how to minimize the impact of the use of implementation dependent features of Ada. Tradeoff analyses were conducted using benchmark programs to determine the execution performance of portable software developed using

workarounds to implementation dependencies.

As a result of these studies, portability guidelines were generated, addressing the following five major concerns:

- Eliminate the need for and use of implementation-dependent features where possible.
- When implementation-dependent features are necessary, provide a translation function between the generic approach to solving the problem and the implementation specific solution. This translation will be unique for each port.
- When translations are not possible, use the most conventional approach possible. Document all implementation dependencies thoroughly.
- A clear design and documentation are critical for portability. Software documentation should include a porting manual.
- Always move application specific hardware that interacts directly with the software to the new host during a port. Modularize all hardware specific software so that if the hardware cannot be moved to the new system, it will be clear which parts of the software will need to be changed.

2.5.2 Overview

The guidelines are organized into nine categories: Erroneous Programs/Order Dependencies, Storage Issues, Performance Issues, Tasking Issues, Interrupt Processing Issues, Numeric Issues, Subprogram Issues, Input/Output Issues, and Other Issues.

In each category the issues are defined and concrete recommendations to achieving portability are given. The recommendations often reference the appropriate sections of the Ada standard for specific semantic information on the construct under discussion. Examples are provided to illustrate significant points.

2.5.3 Strengths

There are two major strengths:

- These guidelines are specific in their focus on real-time issues. The most obvious strength of these guidelines is their treatment of issues omitted by all the other guidelines. Rather than repeat information and recommendations given in other guidelines, the guidelines provide references to the other portability guidelines and go on to cover other issues.
- The guidelines are organized functionally rather than strictly following the Ada standard organization. This allows a software developer to immediately focus on the issues of concern for the application.

2.5.4 Weaknesses

There are two major weaknesses:

- The primary weakness of these guidelines is the minimal use of examples to illustrate the points being made.
- The guidelines provide no information on the risks, costs or benefits associated with using a specific feature in the manner suggested. Unlike the SofTech guidelines that provided the developer with information on the impact of a feature to behavior, source modification, and safety, these guidelines give the developer no details with which to make an informed tradeoff analysis. Furthermore, the guidelines frequently state rules to be followed without giving any justification or explanation.

3. CONCLUSIONS

In reviewing these guides, several conclusions are drawn. These conclusions center around the commonality of programming style, the domain in which portability is applied, and the research into portability guidelines.

Portability is as much a philosophy as it is a practice. For example, program style is a common component among the guides reviewed. While attention to style itself does not guarantee a positive impact on portability, the philosophy of style establishes conventions which when followed may promote portability. The emphasis on programming style is an indication of its importance and strongly suggests that software development efforts that do not establish and enforce programming style conventions will negatively impact portability.

The type of domain affects portability. For example, portability in a real-time, embedded domain would be viewed differently if in a database domain. The rules or guidelines governing development of portable code is identical for both domains, however, the emphasis or use of specific rules may be different.

Finally, the status of research into portability is important. The majority of guides we reviewed were produced shortly after Ada became a standard in 1983. Official DoD interpretations of the Ada standard have occurred since the standard was released none of the guides have been updated to reflect any interpretations affecting portability.

APPENDIX E

IDA PAPER P-2061

FOREWORD

The paper presented in this appendix was prepared during the Spring of 1988 at the beginning of an Ada project undertaken by the Defense Logistics Agency (DLA) in Columbus, Ohio. The Ada project was part of an evolutionary approach to achieve a more open system architecture by which users could have access to the data resources and application software resident on geographically remote computer systems. The primary purpose for the use of Ada was to develop system software that could interface with non-Ada application systems and heterogeneous computer systems, and which could be re-used by adaptation to the client-server model of very different applications. The recommended standards for software configuration management, portability, and coding provided in this paper were offered as a minimal set of standards for the DLA Ada project. These standards were followed by the DLA Team. The Ada software developed during training workshops has been re-used and enhanced since by a small team (less than eight) programmers who have successfully developed a command standard transaction processing system in Ada. Ada has been used as the software platform for user access to information generated by COBOL programs on a host computer remote from the users and for a nation-wide transaction processing system for vendors doing business with DLA.

1. INTRODUCTION

1.1 PURPOSE

The purpose of this paper is to recommend a set of software standards for use by the Defense Logistics Agency. These recommendations are related to the effort by DLA to evaluate the Ada programming language as an Agency standard.

1.2 SCOPE

These recommendations cover three areas of software development: software configuration management (SCM), portability of Ada programs, and Ada coding standards. Recommendations in the first area are independent of any particular programming language. The second area is one in which few standards have yet found widespread acceptance, and the recommendations are only of a very general nature. The third area, Ada coding standards, includes such issues as naming conventions, appropriate and inappropriate statement types, and packaging conventions.

Although the principal scope of this paper is that of software standards and their potential use by DLA, many of its recommendations, principally those concerning coding standards and portability, are derived from lessons learned during the earliest stages of the DLA Ada Prototype Project described in Section 3. Among these experiences were the attempt to erect a workbench of Ada tools and the on-going experiences gained from several intensive Ada training sessions. The workbench experience is documented in IDA Memorandum M-387, *Compiling and Porting the NOSC Tools for Use by the Defense Logistics Agency.*

1.3 BACKGROUND

The Defense Logistics Agency is engaged in a long-term program, the Logistics Systems Modernization Program (LSMP), to modernize its Automated Information Systems. A principal thrust of the LSMP is to determine the feasibility of using the Ada language in DLA applications.

To that end, IDA and DLA are engaged in an Ada Prototype Project. This project involves the creation of an preliminary Ada Programming Support Environment, training a group of DLA programmers in the use of Ada, and the writing of large-scale software projects to demonstrate Ada's capabilities. The functional nature of these demonstrations will be similar to that of the COBOL software currently in use at DLA. Since the project will be distributed through three different machine tiers, an IBM, a Gould, and several Zenith PCs, it will also demonstrate Ada's capacity for portability of software.

One principal requirement for the Prototype Project is an Ada Programming Support Environment (APSE). The APSE is a "workbench" of Ada tools which can later be expanded and modified as the modernization project itself is expanded and modified.

In IDA Memorandum Report M-294, *Ada Prototype Project,* it was recommended that tools for the APSE be acquired by first examining public domain software before resorting to commercially available software. A major source of public domain software is the SIMTEL20 Repository, which contains a large number of available Ada programs. Many of these were selected for potential inclusion in the DLA APSE. Since these tools were commissioned by the Naval Ocean Systems Center, they are commonly referred to as the NOSC tools.

A team of DLA personnel was selected for training in Ada. Their collective expertise is in COBOL, and most of the members are proficient in traditional Automated Data Processing methodology. In addition, two of the team members have considerable backgrounds in systems programming.

The team received six weeks of intensive training in Ada. The classes were given by personnel from TeleSoft, Inc., whose compiler will be used during the Prototype Project and beyond. The textbooks for this course were: *Software Engineering with Ada* and *Reusable Software Components with Ada by Grady Booch* and *Understanding Concurrency in Ada* by Kenneth Shumate.

1.4 APPROACH

There are three different categories of recommendations in this paper, and they derive from numerous sources:

- The trials of erecting the APSE, and the insurmountable difficulty of porting much of it to DLA, were the sources of many of the recommendations concerning configuration management and portability.
- Observing the DLA team's training sessions provided valuable insight into the the type of coding standards and guidelines needed. In addition, many experiences of the authors of some NOSC tools provided other coding recommendations.
- Finally, general experiences gained in using other Ada environments contributed to some of the recommendations in all categories.

Section Two of this paper presents findings and conclusions based on these experiences. Section Three presents recommendations and provides a rationale for each recommendation.

2. FINDINGS AND CONCLUSIONS

Findings in the three areas of SCM, portability, and coding standards are discussed, and conclusions in each area are presented.

2.1 SOFTWARE CONFIGURATION MANAGEMENT

- **Finding 1.** Many of the NOSC tools chosen for the DLA APSE exist in multiple and incompatible versions. This principally applies to the large tools.
- **Finding 2.** The software environment at DLA presently has few tools to aid or enforce SCM. Erection of a dependable system will substantially impact the success of the entire Ada prototype project.
- **Finding 3.** Major components of the SCM system at DLA will need to include controls for multiple versions of source files, mechanisms for standardized software releases, and management of released object code.

2.1.1 Discussion

Of the problems encountered in compiling the NOSC tools, those stemming from poor SCM were the most difficult to solve. Principally, there were variant versions of many packages, due to the fact that the SIMTEL20 sources reflect multiple releases of the tools. The variants existed at different levels of software: some packages had been subsumed into other packages, creating inconsistent dependencies. Others merely reflected earlier and later versions of the same package. These difficulties were especially prominent in large tools comprising many source files.

These problems were overcome with difficulty. It is noteworthy that IDA also had available another version of some NOSC tools, obtained through GTE. While this other source of the tools added some confusion, it was only from the GTE versions that the needed versions of some packages were found.

2.1.2 Conclusions

For a software project of any substance, there is need for a dependable SCM system. Given the nature of Ada, where a basic intention is to achieve a high degree of modularity, effective SCM is even more crucial. An acceptable SCM system will minimally contain a set of protocols and standards for version and revision control, as well as a means to map the correct version of source code to the object code. An acceptable

SCM system will also contain a reliable mechanism for tracking and documenting releases.

Additionally, an Ada project will need a mechanism that governs compilation of several files. This entails determining the correct order of compilation as well as performing the actual invocation of the compiler.

2.2 PORTABILITY

- **Finding 4.** The NOSC toolset contains several tools that were written with non-validated compilers. These tools will not compile with a valid compiler.
- **Finding 5.** The majority of the tools examined contained system-dependent code. In the larger tools, the dependencies were often dispersed through several packages, making them highly difficult to port to other systems.
- **Finding 6.** Even with these dependencies removed, the large NOSC tools could not be compiled using the Gould compiler at DLA. One of the tools was eventually compiled by a considerable reworking of the sources. The executable that was created failed in elaboration.

2.2.1 Discussion

Of the PC-based tools examined, half have been successfully ported to DLA; of the mainframe-sized tools, none. In the case of the failing PC-based tools a major cause was poor or illegal code, due to use of non-validated compilers as development environments. In the case of the mainframe-sized tools, it was due partially to overreliance on VAX® system calls, and especially to a capacity limitation of the DLA Gould compiler. The attempt to compile the NOSC tools is discussed in detail in IDA Memorandum Report M-387, *Compiling and Porting the NOSC Tools for use by the Defense Logistics Agency.*

2.2.2 Conclusions

It is generally misleading to speak of truly "portable" code; such software is relatively rare. The term "portable" more often refers to code that needs only a small amount of alteration in order to compile on different machines; portability is thus a measurement of such alteration. Code which has a high degree of is a portability is code wherein the needed alterations are easily made. Conversely, non-portable code requires complicated alterations, or is written in such a way that the location where alterations are needed is not easily determined.

® VAX is a registered trademark of Digital Equipment Corporation.

It is also apparent that portability may be an issue over which the programmer has little or no control. In the case of the NOSC tools, even though the Gould computer is a reasonably large machine, its compiler was not able to compile code that had compiled without problem on a VAX.

Since one of the stated goals in the DLA Ada project is achieving software portability through three machine tiers, this issue is fundamental; it is one of the prime points that the project is meant to demonstrate. The DLA team must obviously regard portability as an element that must be present at the earliest level of design, and not, as in the NOSC tools, as a consideration after the code has been written. They should also be aware of the limitations of their compiler, and should be urged to design their code accordingly.

2.3 CODING STANDARDS

- **Finding 7.** The coding standard found in the course's textbooks represents only one possible standard for good Ada code.

2.3.1 Discussion

Most of the DLA team members came to Ada from backgrounds in COBOL, and have need of guidelines in writing Ada code. While the definition of an "Ada style" is, to some extent, a matter of opinion, there is is a genuine need for some practical guidance in this area.

At the present, the primary models available to the team members derive from the texts used in the training sessions. In addition, the code for the NOSC tools provide a wide range of code quality and conventions. The code in the textbooks represents a particular kind of coding style. This style is discussed in detail in Section Three. The code in the NOSC tools ranged from unacceptable to excellent; the experiences of some of the NOSC tool authors provided a source for some of the recommendations made later in this paper.

2.3.2 Conclusions

One of the fundamental aims of the Ada language is the writing of maintainable, reusable code. As a means toward that end, it is vital that code be easily readable: in one sense, readability is the hallmark of well-crafted code. Readability in this sense does not refer to documentation, but rather to the actual compilable code. In particular, coding practices that favor dense, abbreviated code are to be avoided.

Coding conventions are, in themselves, major contributors to code readability. Also, since Ada usually involves simultaneous references to multiple source files, any

coding conventions that simplify these references are beneficial; any that hamper it are poor.

3. RECOMMENDATIONS

There are three categories of recommendations: software configuration management (SCM), portability, and coding standards. Five recommendations for SCM, four recommendations for portability, and fifteen recommendations for coding standards are presented.

3.1 Software Configuration Management

1. *Since the Gould computer will be the major development area, configuration management protocols should be governed by the UNIX® operating system.*

The eventual disposition of the Ada project through the three architectural tiers is not yet determined. It is clear, however, that the Gould, using the UNIX operating system, will be central to the project. UNIX also provides a foundation of language-independent tools that partially offset the current absence of a true APSE.

2. *The DLA team should be encouraged to use available SCM tools at every stage of the Ada project.*

There are existing tools that have proven beneficial to SCM. UNIX's *make* and RCS utilities are examples of them. (*make* is a tool that automates compilation of large systems, and RCS is a revision control system for controlling changes to text files.) In addition to encouraging use of tools such as these, other tools, such as mechanisms that automatically generate makefiles, or that facilitate using RCS, should also be developed. Implementation of these mechanisms has already begun.

3. *A mechanism to permit orderly release of source files and executables should be developed.*

The notion of "releasing" software is present when there are several programmers working on interrelated code. There must be an orderly process that allows tested software to be used by others, but that permits a programmer continually to improve it. Such a process depends on many factors: the released version of the source must be accessible; the compiled code must be stored in a safe location, so that other users who rely on it can do so indefinitely; and the mechanism whereby a release is made must be easily invoked so

® UNIX is a registered trademark of AT&T Bell Laboratories.

that it will be used often.

4. *A mechanism that tracks and documents releases should be developed.*

The need for tracking releases is vital. It is often necessary, for instance, to rescind an erroneous release, a process that involves reconstruction of an earlier configuration of the system. Without a tracking mechanism, reconstruction of any particular system configuration is likely to be impossible.

5. *SCM standards for the DLA project should be adhered to by all team members without exception.*

Though notional agreement with this recommendation is probably near universal, experience has shown that SCM standards are those that are followed the least. Experience has also shown that lapses in this area are the most damaging. There are, for instance, costs that can propagate far beyond the awareness of the developers. This point is amply demonstrated by the NOSC tool experience.

One guideline for DLA in selecting its SCM standards is the ANSI/IEEE Std 828-1983, *Software Configuration Management Plans.* It is recommended that the DLA team investigate this document before making any specific decisions in the area of SCM.

3.2 Portability

1. *For each given program, all system interface should be isolated in a single package.*

Software must communicate to the native operating system. In the case of code written for the DEC Ada compiler, for instance, system calls are invoked through a package STARLET, supplied by DEC. On UNIX systems, Pragma Interface(C) or Pragma Interface (UNIX) perform similar roles.

Making such software portable involves isolating this communication in a single location. If the system-dependent code is distributed throughout several packages, then the code will port to another machine only with difficulty.

When the software under consideration involves two or more executables, it is further recommended that each have a separate interface package. This will help avoid a situation encountered in the NOSC tools, where a system interface package was used by several executables. The interface was changed for some, but not all of the executables, resulting in an untenable set of package dependencies. Using a separate package for each executable ensures a necessary independence of executable programs.

2. *Reliance on constants that are system-dependent should be avoided.*

Constants such as those found in package System concern capacities of the host compiler, such as the degree of precision in real numbers. If code depends on a factor like this, then that code is not really portable.

The following code will compile with the DEC Ada compiler, but will fail with some others:

```
package Real_Numbers is
...
type Big is digits 15;
                -- This will fail unless the
                -- value of System.Max_Digits
                -- is at least 15
....
```

One possible alternative is:

```
with System;
package Real_Numbers is
....
type Big is digits System.Max_Digits;
....
```

But this solution is invalid if there is genuine need for the greater precision. In that case, however, the code will always be erroneous on a smaller machine, and is not portable at all.

By contrast, the following code *involves* constants from package System, but does not *depend* on any particular values for them:

```
with Text_IO;
procedure Numbers is
  package TIO renames Text_IO;
    max_size : constant Integer := (Integer'width) - 2;
    number   : Integer;
    dummy    : String (1 .. 100);
    len      : Natural;

                -- this code will work regardless of
                -- the actual size of Max_Int.
begin
  TIO.Get_Line (dummy, len);
  if len <= max_size then
    number := Integer(dummy(1..len));
  else
    TIO.Put_line ("Input value too large");
  ...
```

3. *Excepting generics, source files should contain a single compilation unit. In the case of generics, source files should contain precisely one generic specification and one generic body.*

When a source file contains more than one compilation unit and one of the units fails in compilation, different compilers will behave differently. One possibility is for the compiler to reject the entire compilation; that strategy is used by the Gould compiler. If the source file is very long, with numerous compilation units, and the failure occurs at the very end of the compilation, the wasted time can be considerable.

As a single exception to this recommendation, the Ada Language Reference Manual (ANSI/MIL-STD 1815A) permits an implementation to require generic specifications and bodies to share the same source file. Since the Gould compiler makes this requirement, then generic compilation units should be the only occasion when one source file contains more than one compilation unit. In such cases, the source file should contain no more than two units.

4. *Large arrays, those larger than 1000 elements, should be initialized by slice assignments and not by a single aggregate assignment. If possible, such data structures should be avoided.*

This recommendation stems from the principal reason that the large NOSC tools could not successfully compile at DLA. Several packages in the tools were automatically

generated code, containing large aggregates of integers. These aggregates were initialized by a single assignment statement. In all cases, these packages failed to compile at DLA.

The solution in this case was to break the large aggregate assignment into smaller slice assignment. Thereafter, one such package was successfully compiled. But the executable that was generated failed, and it is has not been determined whether the tools can be brought to successful execution under any conditions.

It seems a safer course to recommend that the Ada style in packaging, i.e., small, modular packages, be brought to bear on data structures as well. Otherwise, as in the NOSC tools, code can be created which will compile successfully on one compiler and not on another, and there may be no possible way to port it because of capacity limitations.

3.3 Coding Standards

This section contains recommendations about naming conventions, packaging conventions, and other coding conventions.

3.3.1 Naming Conventions

1. *The naming conventions that are established should be consistent throughout the entire project, and used by all members.*

This point is self-explanatory, but can not be overemphasized. Even if all of the following recommendations are rejected, there is need for consistent naming conventions across the project.

2. *Whenever practical, use descriptive prefixes for subprograms, especially functions.*

Subprograms are generally entities that "do" things, and the precise nature of what is done should be clear from the subprogram's name. Prefixes like "Is_", "To_", "From_", "Has_" and similar others, can provide this clarity:

"Is_"
- for a function that returns a boolean
- result from making an identity test.

"Has_"
- for a function that returns a boolean
- result from making an attribute test.

"From_",
"To_"
- for a function that converts a value into
- another value. The "From" prefix describes
- the precondition of the function; the "To_"
- prefix describes the postcondition. The choice
- is dependent upon the function's principal work.

While these prefixes deal with functions only, parallel examples for procedures are easily imagined. As an example of the value of descriptive names, consider the lack of clarity in the following specifications:

```
function Lower_Case (item : Character) return Character;
      — This function returns a lower case character
      — from an upper case character.


function Lower (item : Character) return Boolean;
      — This function returns true if a character is
      — in lower case.
```

These might typically be used as follows:

```
c : Character := 'Z';
begin
 ...
   if not Lower(c) then      --??lower than what??
     c := Lower_Case (c);
 ...
```

A preferable, though more verbose, alternative is both self-documenting and consistent:

```
function To_Lower_Case (item : Character) return Character;
function Is_Lower_Case (item : Character) return Boolean;

...
  c : Character := 'Z';
  begin

...

        -- conventional use of "is_"
        -- indicates a Boolean test
    if not Is_Lower_Case(c) then
      c := To_Lower_Case (c);
...
```

Finally, if the appearance of "... not Is_Lower_Case" is offensive, then the following addition:

```
function Is_Upper_Case (item : Character) return Boolean;
```

leads to:

```
    if Is_Upper_Case(c) then
      c := To_Lower_Case (c);
...
```

3. *Abbreviations should not be used in subprogram names. Wherever practical, subprogram names should be entirely self-documenting.*

The semantic content of abbreviations is a highly subjective matter. While such specifications as:

```
procedure Val (item : Item_Type);
```

will probably connote "Value" to most people, it is quite possible that

```
function Mat_mpy (mat_1, mat_2 : Mat_Type) return Mat_Type;
```

will not be meaningful except to its author. Changing this to

```
function Matrix_Multiply (
        matrix_1,
        matrix_2 : Matrix_Type) return Matrix_Type;
```

results in a considerable increase in readability.

4. *Naming conventions should be chosen so as to avoid unreadable code.*

There are many viewpoints on good naming conventions, especially as regards names of types and objects. The DLA Ada team used texts by Booch and Shumate. Particularly in reference to the Booch texts, the DLA team should be made aware of some different points of view.

There are several objections that can be made to the Booch style. First, since all objects begin with the four characters "The_", there is an unwelcome element of sameness to each object. And if there are several objects being manipulated in the code, or several fields of the same record object, the result can be extremely awkward to read. The following is an example:

```
if The_Ring.The_Back = 0 then
  raise Underflow;
elsif The_Ring.The_Back = 1 then
  The_Ring.The_Top := 0;
  The_Ring.The_Back := 0;
  The_Ring.The_Mark := 0;
else
  The_Ring.The_Items(The_Ring.The_Top..The_Ring.The_Back-1):=
  The_Ring.The_Items(The_Ring.The_Top + 1)..The_Ring.The_Back);
  The_Ring.The_Back := The_Ring.The_Back - 1;
  if The_Ring.The_Mark > The_Ring.The_Top then
   The_Ring.The_Mark := The_Ring.The_Mark - 1;
end if;
```

(Booch, p.185)

Another weakness in the above convention is that when two objects of the same type are used, they are distinguished by prepositions, commonly "To" and "From". But the use of these is incorrect as regards common English meaning. As an example:

```
for Index in From_The_Map.The_Items'Range loop
   if From_The_Map.The_Items(Index).The_State = Bound then
      Find (From_The_Map.The_Items(Index).The_Domain,
         To_The_Map, The_Bucket);
```

(Booch, p.230)

The intended meaning here is to distinguish between a "to" map and a "from" map, one a source and one a destination. But in simple English, using "...to the x...from the x..." commonly refers to the same "x". To be consistent with English usage, it would need to read: "The_To_Map ... The_From_Map", at which point common sense rebels.

5. *Wherever practical, use descriptive suffixes to denote common data types.*

It is undoubtedly a good practice to separate type names from variable names; that is an obvious intent of the conventions discussed in Recommendation 4. But a better way to achieve that goal is to place a descriptive suffix on the type, rather than an article on each variable. By using such suffixes as:

"_ptr" – for access types.
"_rec" – for record types.
"_arr" – for array types.
"_type" – for enumeration types.

the following code:

```
type Color_Type is (red, white, blue);
type Color_Ptr is access Color_Type;
color : Color_Ptr;
...
color := new Color_Type'(red);
if color.all /= blue then
...
```

will be both clear and succinct. It should be noted that using abbreviated *suffixes* on types, unlike abbreviations for the *nouns* or *verbs* in subprogram names (cf. Recommendation 3) is an acceptable practice, since suffixes indicate only typical data types such as arrays, records, and pointers.

6. *Use simple names (without prefix or suffix) to denote variables.*

Generally, type names are used once, variables names several times. The descriptive prefix or suffix should be used at the point where the type needs to be discerned, nowhere else. Of the following two examples, the first is preferable:

```
type Node;
type Node_Ptr is access Node;
type Queue_Rec is
record
  Front : Node_Ptr;
  Back  : Node_Ptr;
end record;
...

procedure Copy (
  From : in Queue_Rec;
  To   : in out Queue_Rec) is
...
...
  if From.Front = null then
    To.Front := null;
    To.Back := null;

...
```

```
type Node;
type Structure is access Node;
type Queue is
record
  The_Front : Structure;
  The_Back  : Structure;
end record;
...

procedure Copy (
  From_The_Queue : in Queue;
  To_The_Queue   : in out Queue) is
        ...
        ...
          if From_The_Queue.The_Front = null then
            To_The_Queue.The_Front := null;
```

```
To_The_Queue.The_Back := null;
...
```

(Booch, p.149)

3.3.2 Packaging Conventions

1. *Subprograms, whether functions or procedure, should be brief; each should accomplish a single action.*

One of the hallmarks of the Ada style is a high degree of modularization, with the restriction of a subprogram to a single action. The benefits of this are twofold: first, since the subprogram has only one effect, it can subsequently be reused in a variety of contexts. Second, the code of such a subprogram will necessarily prevent the dense, unmanageable code that Ada was intended to avoid.

As a simple means to achieve this goal, it is further recommended that a typical subprogram be restricted to a very few lines of code. An upward limit is difficult to determine, but a subprogram that is larger than fifty lines is probably too long.

Some subprograms will perforce violate this recommendation; sometimes such things as a very lengthy case statement are the best solution to a particular problem. But in the general order, this recommendation can restrict these occasions to a minimum, and can also enforce a logical rigor conducive to good software engineering practice.

2. *Wherever possible, subprograms should have no side effects. All effects of a subprogram should be centered on parameters.*

The principal way that a subprogram can have a side effect is by acting on global variables. Global variables are generally avoided by modern software engineering practices, and an Ada programming style should generally follow this practice.

3. *The number of parameters for subprograms should rarely if ever exceed six.*

This is related to recommendation 1 concerning single-action subprograms. If a subprogram genuinely has need of many parameters, it is worth considering whether the chosen data structures are appropriate. A common possibility is that the several parameters can be collected into a single record type, and passed in as a single parameter. If this is not appropriate it is then worth considering if the action of the subprogram is itself appropriate, or whether the subprogram is really doing the work of several procedures.

4. *Wherever possible, variables should not appear in package specifications. Variables whose life span must exceed a given subprogram call should lie in package bodies.*

The presence of variables in specifications is closely related to recommendation 2 concerning the danger of side effects. A variable in a specification is vulnerable to all units that 'with' the package. The package body is the appropriate location for variables whose life span must exceed a given subprogram call, since only the package's own subprograms may alter such variables.

5. *Constants in package specifications should be replaced by parameterless functions.*

The presence of a constant in a specification is always subject to the danger that the constant will need to be changed and the package recompiled, thus rendering all dependent units obsolete. The effect of a visible constant can be gained without this risk by using a parameterless function to return the constant value. The second version below is preferable to the first:

```
package Data is
 ....
Int_Value : constant Integer := 100;
 ....
```

```
package Data is
 .....
function Int_Value return Integer;
 .....
```

The actual integer value is then located in the package body, which can be altered and recompiled with no other dependencies involved. Note that any calling program that uses this value does so with precisely the same code for both versions:

```
with Data;
procedure Do_Something is
 ...
x := Data.Int_Value;
```

6. *Wherever possible, avoid subunits.*

Most of the asserted benefits of subunits are imaginary. Though textbook examples of development, where a body is stubbed out and the subunits developed one by one, look quite reasonable, experiences by many Ada programmers suggest that such neat sequences of development seldom occur.

This recommendation is potentially controversial, since textbooks generally urge the frequent use of subunits. But it is the author's experience in various Ada projects that programmers in large numbers come to avoid subunits except in the most exceptional circumstances.

3.3.3 Other Coding Conventions

1. *Avoid unnecessary WITH clauses in specifications.*

It is not uncommon to include a WITH clause in a package specification even if the 'withed' unit is not referenced until the body. Except for the predefined units such as Text_IO, this practice can have unfortunate results. Principally, it will add unnecessary dependencies, which in turn will trigger unnecessary recompilations throughout the development phase. In addition, such a practice is a mark of poor engineering standards.

2. *Use USE clauses seldom if at all.*

The principal objection to the USE clause is that it obscures the location of declarations from the reader of the code. Using USE is not the same as 'information hiding': on the contrary, USE hides valuable information from a person who might desperately need the information that is hidden.

There are only two reasons that USE clauses might be justified:

a. *To avoid cumbersome code filled with dot-selected identifiers.*
b. *To gain visibility of equality and inequality.*

In the first case, it is often a better practice to use package renames, which simplify the appearance of the code and still allow the reader to locate references. As an example, the second fragment below is preferable to the first:

```
with A_Types. B_Types, C_Types;
use A_Types, B_Types, C_Types;
package Data is
   var_1 : Color := gray;
   var_2 : Shade := Initialize;
   var_3 : Hue := Initialize (var_1);
   var_4 : Hue := Initialize;
```

```
with A_Types, B_Types, C_Types;
package Data is
   package A renames A_Types;
   package B renames B_Types;
   package C renames C_Types;

   var_1 : A.Color := A.gray;
   var_2 : B.Shade := B.Initialize;
   var_3 : C.Hue := C.Initialize (var_1);
   var_4 : C.Hue := B.Initialize;
```

It is also worth noting that the USE version obscures the fact that var_3 and var_4 are initialized by functions in different packages, a point that is explicit in the second version.

The second reason to add a USE clause is to gain the visibility of the equality operator. In such cases, the following are possible alternatives:

a. If the equality visibility is needed only once, then the "=" notation is not a terrible inconvenience.

```
with Data_Types;
package body Something is
  package DT renames Data_Types;
...
  procedure Do_Something is
     x : DT.AnyKind;
  begin
    x := Some_Function;

...
                -- this is the only time
                -- the "=" is needed
          if DT."=" (x,DT.red) then ...
```

b. If the visibility is only needed within a single procedure, then the USE clause can also be located there, as in the following example:

```
with Data_Types; package body Something is
...
  procedure Do_Something is
        x : Data_Types.AnyKind;

                -- AnyKind is defined -- in package Data_Types
      use Data_Types;
                -- USE clause is in effect -- only within this
                procedure
      begin
        x := Some_Function; if x = red then ...
```

Note also that the USE clause appears only after the declaration of variables: the location of type 'AnyKind' is not hidden by USE.

3. *Exceptions should be used only for true run-time error conditions. They should not be used for recovering from expected conditions.*

In most compiler implementations, exceptions have a high overhead. Further, the intended use of exceptions in the design of Ada was not to include any message-passing functionality, but only to provide a means to recover from runtime errors.

For instance, consider the following:

```
function Calculate (x : Integer) return Integer is
begin
 .... -- do some useful computation with x
   return x;
exception
   when Constraint_Error => return 10_000_000;
                -- set x to 10_000_000 whenever the
                -- computation exceeds Max_Int
end Calculate;
```

Code such as this is using the exception handling mechanism of Ada to test boundary conditions of the in parameter. This is the type of test that might better be made in the code instead, if at all possible:

```
function Calculate (x : Integer) return Integer is
   subtype Acceptable_Range is Integer range <some acceptable range>;

begin
   if not (x in Acceptable_Range) then
         return 10_000_000;
   else
    .... -- do some useful computation with x
   end if;
   return x;
end Calculate;
```

4. SUMMARY

The standards enumerated in this paper are based on lessons learned when COBOL programmers at the Defense Logistics Agency were making a transition to Ada. There is no intention to cover all possible areas, but rather to focus on the standards most commonly needed by experienced programmers making such a transition. These standards should therefore be regarded as a starting point, over which a fuller set of agency-wide standards can be erected. The full complement of DLA software standards can and should be perceived as being a major contribution by the Ada Prototype Project to the eventual success of DLA's Logistics Systems Modernization Plan.

The matter of standards should not be thought of as "elementary", or an issue for novices only. The need for consistent, sensible standards in modern software engineering is indisputable. Especially given the probable scope of projects written in Ada, there can be little doubt that ad hoc, on-the-spot conventions and standards will be detrimental factors in any project's success. From both an engineering and a management viewpoint, the more a project is bound to a uniform, common-sense set of software standards, the more the members of that project are free to focus their energy on the real problems - designs, algorithms, optimizations, abstractions - that face software engineering.

BIBLIOGRAPHY

Gardner, M.R., R.L. Hutchison, & T.P Reagan. 1986. *A portability study based on rehosting WIS Ada tools to several environments.* Mclean, VA: The MITRE Corporation.

Nissen, J.C.D. & Peter J.L. Wallis. 1984. *Portability and style in Ada.* Cambridge: Cambridge University Press.

SofTech, Inc. 1984. *Ada portability guidelines.* Waltham, MA: SofTech, Inc.

Tracz, Will. 1987. *Ada reusability efforts: A survey of the state of the practice.* Stanford, CA: Computer Systems Laboratory, Stanford University,

Distribution List for IDA Paper P-2456

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Mr. James Robinette JIEO/TVCF Defense Information Systems Agency Center for C3 Systems 3701 N. Fairfax Dr. Arlington, VA 22203	5
Other	
Mr. Terry Courtwright STARS Technology Center Suite 317 1801 N. Randolph St., Suite 400 Arlington, VA 22203	1
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dr. James P. Pennell AT&T Room 2025 8065 Leesburg Pike Vienna, VA 22182	
Mr. Karl H. Shingler Department of the Air Force Software Engineering Institute Joint Program Office (ESD) Carnegie Mellon University Pittsburgh, PA 15213-3890	1
IDA	
General Larry D. Welch, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Ms. Ruth L. Greenstein, HQ	1
Dr. James R. Carlson, SED	1
Dr. Robert P. Walker, SED	1
Dr. Kevin J. Saeger, SED	1

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Cy D. Ardoin, CSED	1
Mr. James Baldo, CSED	1
Mr. John M. Boone, CSED	1
Mr. Bill R. Brykczynski, CSED	1
Ms. Anne Douville, CSED	1
Dr. Dennis W. Fife, CSED	1
Dr. Karen D. Gordon, CSED	1
Ms. Audrey A. Hook, CSED	1
Ms. Deborah Heystek, CSED	1
Dr. Richard J. Ivanetich, CSED	1
Mr. Robert J. Knapper, CSED	1
Mr. Terry Mayfield, CSED	1
Ms. Katydean Price, CSED	5
Ms. Beth Springsteen, CSED	1
Dr. Richard L. Wexelblat, CSED	1
Dr. Craig A. Will, CSED	1
IDA Control & Distribution Vault	3